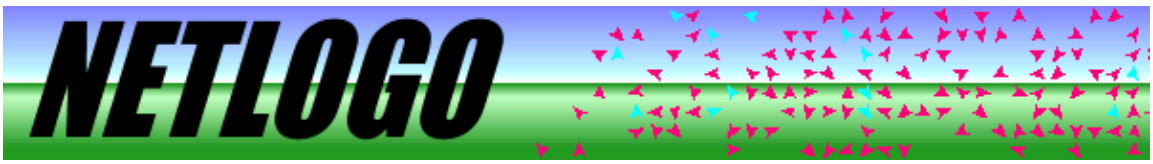


NetLogo 1.1 User Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of independent "agents" all operating in parallel. This makes it possible to explore the connection between the micro−level behavior of individuals and the macro−level patterns that emerge from the interaction of many individuals.

NetLogo lets students open simulations and "play" with them, exploring their behavior under various conditions. It is also an "authoring tool" which enables students, teachers and curriculum developers to create their own models. NetLogo is simple enough that students and teachers can easily run simulations or even build their own. And, it is advanced enough to serve as a powerful tool for researchers in many fields.

NetLogo has extensive documentation and tutorials for all of its features. It also comes with a Models Library, which is a large collection of pre−written simulations that can be used and modified. These simulations address many content areas in the natural and social sciences, including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. Several model−based inquiry curricula using NetLogo are currently under development.

NetLogo can also power a classroom participatory−simulation tool called HubNet. Through the use of handheld devices such as Texas Instruments (TI−83+) calculators, each student can control an agent in a simulation over a wireless network. (Note: HubNet is not yet released to the general public. Follow this link for more information.)

NetLogo is the next generation of the series of multi−agent modeling languages that started with StarLogo. It builds off the functionality of our product StarLogoT and adds significant new features and a redesigned language and user interface. NetLogo is written in Java so it can run on any operating system (Mac, Windows, Linux, et al). It can be run as a standalone application, or as an "applet" within a web browser.

Features

You can use the chart below to help familiarize yourself with the new features NetLogo has to offer.

StarLogoT	NetLogo	Features
X	X	Fully programmable
X	X	Language is Logo dialect extended to support agents and parallelism
X	X	Interface builder w/ buttons, sliders, monitors, switches, text boxes
X	X	Info area for annotating your model
X	X	Powerful and flexible plotting system
X	X	Agent Monitors for inspecting agents
X	X	Export and import model function (save and restore state of model)
	X	Cross−platform: runs on MacOS, Windows, Linux, et al
	X	Web−enabled (run within a web browser or download and run locally)

	X	Models can be saved as applets to be embedded in a web page
	X	Unlimited numbers of agents and variables
	X	Double precision arithmetic
	X	Simplified language structure
	X	"Agentsets" make many programming tasks easier
	X	Syntax−highlighting code editor
	X	Rotatable and scalable vector turtle shapes
	X	Exact on−screen turtle positioning
	X	Redesigned user interface
	X	Text labels for turtles and patches
	X	New primitives
	X	BehaviorSpace: a tool used to collect data from multiple runs of a model
	X	HubNet: participatory simulations using networked handheld devices
	X	Converts StarLogoT models into NetLogo models

What's New?

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

Version 1.1 Rev D (August 2002)

- content:
  - fixed links in PDF version of User Manual that took you to the wrong page
  - the "Known Issues" section of the User Manual now lists a few more known bugs
  - improved models: Rabbits Grass Weeds (better default slider settings), Painted Desert Challenge (better default slider settings), CA 1D suite (faster, clearer code)
  - adjusted the interfaces of many models so that texts don't get cut off even on Java VMs with a large default font size
- engine fixes:
  - fixed slight periodic bias in random number generator when generating small integers
  - several fixes to `in-radius`: it now wraps around the edges, and handles radii larger than screen-edge-x/y properly, and no longer leaves out some turtles near the edge of the circle
  - fixed case where the compiler sometimes incorrectly interpreted parentheses inside brackets
  - fixed bug where turtle/patch labels in the bottom row of patches could cause Java exceptions while zooming
  - using `shade-of?` no longer causes an agent's turn to end
  - fixed several bugs in StarLogoT model converter
- interface fixes:
  - fixed bug where changes to the patch size weren't always saved with the model
  - fixed some cases where the Errors tab didn't always highlight the location of syntax errors correctly
  - fixed a bug where the Procedures menu could produce Java exceptions
  - printing the procedures and info tabs now uses a smaller font size
  - fixed Mac-only bug where the "Import Shapes" feature sometimes behaved strangely when choosing a model
  - fixed Linux/Unix only bug where trying to create or edit a plot made NetLogo grind nearly to a halt
  - fixed bug on Java 1.4 VM's where the Tab key didn't work in the Command Center and Procedures tab
  - now on Macs subsequent Java exceptions after the first are no longer lost
  - when running under Classic on OS X 10.2, the "User Manual" menu item on the Help menu now works, and launches your default browser instead of always Internet Explorer
  - when running native on OS X 10.2 (which is not officially supported yet), the Quit menu item now asks you if you want to save your changes, like it should
- fixes to "Save as Applet" feature:
  - improved instructions in saved applets, and made them visible HTML text instead of invisible HTML comments
  - saved applets now bring up runtime error dialogs on all systems when something goes wrong

- saved applets now work when run from the local hard drive (as opposed to a web server), even on Mac OS X, and even on Windows if the model name has spaces in it

Version 1.1 (July 2002)

- content:
  - added "Frequently Asked Questions" section to User Manual
  - new suite of 1−dimensional cellular automata models: Cellular Automata 1D (an improved version of our old "Cellular Automata" model), CA 1D Totalistic, CA 1D Rule 30, CA 1D Rule 30 Turtle, CA 1D Rule 90, CA 1D Rule 110, CA 1D Rule 250
  - new model conversions: Koch Curve, Rumor Mill, Random Walk Left Right
  - new chemistry models: Crystallization Moving, Crystallization Directed
  - improved models: Crystallization Basic (formerly Solidification), Mouse Example (expanded), Tetris (bugfixes), Pac−Man (bugfixes), AIDS (bugfix, cleanup, clearer code), Traffic 2 Lanes (cleanup, clearer code)
- features:
  - "Save as Applet" (on the File menu) lets you embed your model in any web page
  - you can now print the contents of any tab on a printer
  - Procedures tab now has popup menu that lets you instantly jump to any procedure in your code
  - Interface tab is now scrollable
  - Interface tab now has "contextual" popup menus that let you create, edit, and delete interface elements, as well as open turtle and patch monitors; on Macs, control−click to bring up the menus, on other platforms, use the right mouse button
  - agent monitors are now their own floating windows; agent command centers now do syntax highlighting and support command history
  - code editor now auto−indents a new line to the same level as the line above it; arrow keys in combination with option key (Mac) or control key (other OS's) now moves word by word
  - tab key now cycles the Command Center between Observer, Turtles, and Patches
  - sliders now have an optional "units" field
  - performance improvements made so models run faster (most models only a little faster, but some substantially)
  - new turtle graphics options (note: using these options slows down most models a lot; we hope to improve this in a future version):
    - new turtle variable called `size`; edit the Graphics Window and check the "Turtle Sizes" checkbox to enable scaling of turtles according to this variable
    - new "Exact Turtle Positions" checkbox; edit the Graphics Window and check this box to enable drawing of turtles in their exact positions, instead of always in the center of their patch as is the default
- language changes:
  - new primitive `without−interruption` lets you mark a section of code as being executed all at once without any other agents being able to interrupt
  - new primitives `neighbors` and `neighbors4` return an agentset of a turtle's or patch's eight (or four) neighboring patches
  - new primitive `inspect` lets you open an agent monitor for any turtle or patch
  - new primitives `user−choice` and `user−input` for accepting input from user
  - new primitive `read−from−string` for converting strings (e.g. user input or input from a file) into data

- the `and` and `or` primitives now "short circuit", which means they no longer evaluate their right hand arguments unless necessary
- `message` primitive renamed to `user-message`

- engine fixes:
  - fixed Windows-only bug where models that used `plot-pen-reset` a lot tended to crash NetLogo
  - the `random` primitive now works properly on negative integers and on zero
  - fixed bug in `cct`, `hatch`, and `sprout` where sometimes the new turtle was not initialized immediately
  - fixed bug where setting a turtle's x or y coordinate to an extremely large value could cause NetLogo to stop responding
  - fixed bug where editing `screen-edge-x/y` didn't empty the breed agentsets
  - the `turtle` and `patch` primitives now signal a runtime error if passed non-integer arguments
  - fixed compiler bug that could cause a confusing error message if you used an observer-only primitive in the same context as a turtle/patch-only primitive
  - fixed bug where sometimes patches containing only hidden turtles were drawn with an incorrect color
  - fixed bug where changing a label's color didn't always redraw the whole label immediately
  - using the double quote character (") in a plot pen name no longer confuses NetLogo
  - fixed rare bug where a runtime error could cause Java exceptions if it occurred inside a procedure where an agentset was stored in a local variable
- interface fixes:
  - now when you open a widget editor the contents of the default field are selected, and when you use the Tab key the contents of the next field become selected
  - fixed bug where occasionally button presses were ignored
  - now if you try to open a file that is not a valid NetLogo model, you get a warning instead of it just failing silently
  - fixed bug where monitors set to a very high number of decimal places (such as 20) sometimes had a slight rounding error
  - numerous fixes to sliders including:
    - fixed rounding errors where sliders could return slightly incorrect values (that differed from the correct values in the 15th decimal place or so)
    - fixed handling of various unusual values or combinations of values for minimum, maximum, and increment
    - appearance when zoomed is improved
- BehaviorSpace improvements:
  - added an "Export Plot" button and an "Export Behavior Data" button
  - improved behavior of plots with respect to using consistent axes across model runs
  - assorted small usability improvements
- HubNet improvements:
  - all six HubNet activities revised and improved
  - no more TI-83+ button -- login is initiated instead by the new `hubnet-reset` primitive
  - HubNet now supports receiving non-list data types including numbers, strings, and matrices
  - HubNet Programming Guide revised and expanded
  - all old HubNet primitives removed from language and replaced with new set of primitives: `hubnet-reset`, `hubnet-message-waiting?`, `hubnet-fetch-message`, `hubnet-message`, `hubnet-message-source`,

`hubnet-message-tag`, `hubnet-broadcast`, `hubnet-set-tags`, `hubnet-set-client-interface`

Version 1.0 (April 2002)

- User Manual redesigned and expanded
- permanent plot pens
- miscellaneous fixes and improvements

Beta 10 (March 2002)

- models using turtle and patch labels run faster
- improved compiler and runtime errors and error handling
- usability improvements to Command Center
- full string processing capabilities
- plotting modes: line, bar, and point
- miscellaneous fixes and improvements

Beta 9 (December 2001)

- models run faster now
- syntax-highlighting editor
- new language support for working with agents individually
- "Import World"
- integer sliders
- Edit menu
- miscellaneous fixes and improvements

Beta 8 (October 2001)

- miscellaneous fixes and improvements

Beta 7 (September 2001)

- models that use turtle shapes now run much faster
- much larger numbers of agents now supported
- more colorful & attractive interface
- runtime errors
- "Zoom In" and "Zoom Out" for easy viewing
- histograms
- breed-specific turtle variables
- patch labels
- true integer arithmetic
- `random-seed` primitive for repeatable model runs
- miscellaneous fixes and improvements

Beta 6 (August 2001)

- Mac OS X support (in Classic environment)
- Models Library organized into categories, with screen shots
- language now supports turtle "breeds"
- BehaviorSpace, a tool for exploring the "space" of different ways a model can behave
- "Export World", "Export Plot", and "Export Output"
- local variables (`locals` keyword)
- "Find" and "Find Again" buttons in Procedures window
- HubNet can now send data to the teacher calculator
- miscellaneous fixes and improvements

Beta 5 (June 2001)

- models run much faster now
- collection of "Code Examples" added to Models Library
- agent monitors much improved
- miscellaneous fixes and improvements

Beta 4 (March 2001)

- greatly improved general stability & reliability
- web−based installer automates installation process
- runs as stand−alone application on all platforms
- shapes editor for turtle shapes
- miscellaneous fixes and improvements

Beta 3a (December 2000)

- miscellaneous fixes and improvements

Beta 2 (October 2000)

- miscellaneous fixes and improvements

Beta 1 (September 2000)

- first public beta release

System Requirements

NetLogo is designed to run almost any type of computer, but you may experience problems with older, less powerful systems or older versions of system software. The exact requirements are summarized below. If you have any trouble with NetLogo not working on your system, we would like to offer assistance. Please write bugs@ccl.northwestern.edu.

Application or Applet?

There are two ways to run NetLogo:

- Download application. This enables you to run NetLogo as a normal application. (Size of download: usually between 8 and 14 megabytes, depending on options.)
- Run applet on the web within your browser window.

Running on the web is convenient, but downloading the application has some significant advantages:

- Fewer compatibility issues with various operating systems and browsers.
- Starts up faster and models run faster.
- Window is resizable.
- Edit menu is available.
- Keyboard shortcuts for menu items are available.

System Requirements: Application

On any type of computer, a Java Virtual Machine that supports Java version 1.1 or later is required. On the Windows, MacOS, and Linux platforms, you can choose to include the Java Virtual Machine with the download.

On all systems, approximately 15MB of free hard drive space is required.

Windows

- Windows 95, NT, 98, ME, 2000, or XP
- 64 MB RAM

Macintosh: OS X

- OS version 10.1 or later
- 128 MB RAM

Macintosh: OS 8.x or 9.x

- PowerPC processor
- OS version 8.1 or later; 8.5 or later recommended
- 64 MB RAM (96 MB RAM recommended)

Other platforms

NetLogo should work on any platform on which a Java Virtual Machine, version 1.1 or later, is available. If you have trouble, please contact us (see above).

System Requirements: Applet

Your web browser must support Java. In addition, you must have Java enabled in the browser's preferences or settings.

The applet has been tested to work in the following environments:

Windows (95, NT, 98, ME, 2000, XP)

- Microsoft Internet Explorer, minimum version 5
- Netscape, minimum version 6

Macintosh: OS X

- Mac OS X 10.1.3 version or higher required
- Internet Explorer browser 5.1 or higher works
- Netscape and Mozilla may work if you use MRJPluginCarbon
- other browsers not tested

Macintosh: OS 8.x or 9.x

- JVM: Apple's Macintosh Runtime for Java: MRJ 2.2.5 (2.2.3 and 2.2.4 will probably also work)
- Microsoft Internet Explorer, minimum version 4.5 (but version 5 is recommended, and for HubNet users it is required)
- Netscape, minimum version 6

Other Java−enabled OS (Linux, Solaris, et al)

We have not tested this option ourselves, but some users have reported success. If if it doesn't work for you, please let us know. If it doesn't work, you may also try downloading the application instead.

Known Issues

If NetLogo malfunctions, please send us a bug report. See the "Contact Us" section for instructions.

Known bugs (all systems)

- The "Turtle Sizes" and "Exact Turtle Positions" features can cause display anomalies under some circumstances
- Integers in NetLogo cannot be larger than +/− 2 ^ 31, and if you exceed this range, instead of a runtime error occurring, you get incorrect results
- On some Java virtual machines, when you print the Interface tab, there may be pieces missing and/or extra artifacts
- Out−of−memory conditions are not handled gracefully

Macintosh−only bugs

- The "Help" menu has an extra blank item in it
- When running in Classic under versions of Mac OS X prior to 10.2, the "User Manual" item on the Help menu will launch a Classic browser instead of a native browser unless a native browser is already running

Linux/UNIX−only bugs

- Resizing and moving of items in the Interface tab may function erratically on some systems
- Saved applets may not work correctly in Netscape version 4

Planned features

This is a partial list of features we are working on currently. We plan to continue to improve NetLogo. Periodically, new versions will be available from our web site.

- Java API for user−defined extensions to NetLogo
- Improved engine so models run faster
- Optional randomized agent scheduler
- Adjusting the execution speed of a model as it runs (without adding a slider and modifying your code)
- Ability to have multiple models open
- Making a QuickTime movie of your model
- Improved support for HubNet
- Native Mac OS X support (note: we already run well under OS X in Classic mode)

Unimplemented StarLogoT primitives

The following StarLogoT primitives are not available in NetLogo. (Note that many StarLogoT primitives, such as `count−turtles−with`, are intentionally not included in this list because NetLogo allows for the same functionality with the new agentset syntax.)

- `ifelse-report`
- `maxint`, `minint`, `maxnum`, `minnum`
- `import-turtles`, `import-patches`, `import-turtles-and-patches` (note that NetLogo adds `import-world`, though)
- `beep`, `get-date-and-time`, `readlist`, `yes-or-no`
- miscellaneous seldom-used plotting reporters such as `plot-pencolor`, `pp-plotlist`, `pp-plotpointlist`, `ppinterval`, `ppxcor`, `ppycor`, etc.
- `bit`, `bitand`, `bitneg`, `bitor`, `bitset`, `bitstring`, `bitxor`, `make-bitarray`, `rotate-left`, `rotate-right`, `shift-left`, `shift-right`
- `random-exponential`, `random-exponential-list`, `random-normal`, `random-normal-list`, `random-poisson`, `random-poisson-list`, `random-uniform`, `random-uniform-list`, `distribute-random-exponential`, `distribute-random-poisson`
- `close-movie`, `open-movie/setup-movie`, `movie-snapshot`, `snapshot`
- `load-pict`, `save-pict`
- `camera-brightness`, `camera-click`, `camera-init`, `camera-set-brightness`
- `choose-directory-dialog`, `choose-file-dialog`, `choose-new-file-dialog`, `netlogo-directory`, `project-directory`, `project-name`, `project-pathname`, `save-project`

Contacting Us

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you.

Website

Our website at ccl.northwestern.edu includes our mailing address and phone number. It also has information about our staff and our various research activities.

Feedback, Questions, Etc.

If you have general feedback, suggestions, or questions, write to feedback@ccl.northwestern.edu.

Reporting Bugs

If you would like to report a bug that you find in NetLogo, write to bugs@ccl.northwestern.edu. When submitting a bug report, please try to include as much of the following information as possible:

- A complete description of the problem and how it occurred.
- The NetLogo model or code you are having trouble with. If possible, attach a complete model.
- Your system information: NetLogo version, OS version, and so on. This information is available from NetLogo's "About NetLogo" menu item.
- Any error messages that were displayed.

Sample Model: Party

This activity is designed to get you thinking about computer modeling and how you can use it. It also gives you some insight into the NetLogo software. We encourage beginning users to start with this activity.

At a Party

Have you ever been at a party and noticed how people cluster in groups? You may have also noticed that people do not stay within one group, but move throughout the party. As individuals move around the party, the groups change. If you watched these changes over time, you would notice patterns forming.

For example, in social settings, people tend to exhibit different behavior than when they are at work or home. Individuals who are confident within their work environment may become shy and timid at a social gathering. And others who are quiet and reserved at work may be the "party starter" with friends.

The patterns may also depend on what kind of gathering it is. In some settings, people are trained to organize themselves into mixed groups; for example, party games or school−like activities. But in a non−structured atmosphere, people tend to group in a more random manner.

Is there any type of pattern to this kind of grouping?

Let's take a closer look at this question by using the computer to model human behavior at a party. NetLogo's "Party" model looks specifically at the question of grouping by gender at parties: why do groups tend to form that are mostly men, or mostly women?

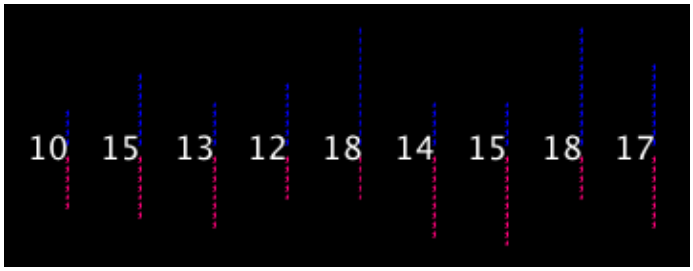
Let's use NetLogo to explore this question.

What to do:

1. Start NetLogo.
2. Choose "Models Library" from the File menu.

3. Open the "Social Science" folder.
4. Click on the model called "Party".
5. Press the "open" button.
6. Wait for the model to finish loading
7. (optional) Make the NetLogo window bigger so you can see everything.
8. Press the "setup" button.

In the Graphics Window, you will see pink and blue lines with numbers:



These lines represent mingling groups at a party. Men are represented in blue, women in pink. The numbers are the total number of people in each group.

Do all the groups have about the same number of each sex?

Let's say you are having a party and invited 150 people. You are wondering how people will gather together. Suppose 10 groups form at the party.

How do you think they will group?

Instead of asking 150 of your closest friends to gather and randomly group, let's have the computer simulate this situation for us.

What to do:

1. Press the "go" button.
2. Observe the movement of people until the model stops.

Now how many people are in each group?

Originally, you may have thought 150 people splitting into 10 groups, would result in about 15 people in each group. From the model, we see that people did not divide up evenly into the 10 groups −− instead, some groups became very small, whereas other groups became very large. Also, the party changed over time from all mixed groups of men and women to all single−sex groups.

What could explain this?

There are lots of possible answers to this question about what happens at real parties. The designer of this simulation thought that groups at parties don't just form randomly. The groups are determined by how the individuals at the party behave. The designer chose to focus on a particular variable, called "tolerance":

Tolerance is defined here as the percentage of people of the opposite sex an individual is "comfortable" with. If the individual is in a group that has a higher percentage of people of the opposite sex than their tolerance allows, then they become "uncomfortable" and leave the group to find another group.

For example, if the tolerance level is set at 25%, then males are only "comfortable" in groups that are less than 25% female, and females are only "comfortable" in groups that are less than 25% male.

As individuals become "uncomfortable" and leave groups, they move into new groups, which may cause some people in that group to become "uncomfortable" in turn. This chain reaction continues until everyone at the party is "comfortable" in their group.

Note that in the model, "tolerance" is not fixed. You, the user, can use the tolerance "slider" to try different tolerance percentages and see what the outcome is when you start the model over again.

How to start over:

1. If the "go" button is pressed (black), then the model is still running. Press the button again to stop it.
2. Adjust the "tolerance" slider to a new value by dragging its red handle.
3. Press the "setup" button to reset the model.
4. Press the "go" button to start running again.

Challenge

As the host of the party, you would like to see both men and women mingling within the groups. Adjust the tolerance slider on the side of the Graphics Window to get all groups to be mixed as an end result.

To make sure all groups of 10 have both sexes, at what level should we set the tolerance?

Test your predictions on the model.

Can you see any other factors or variables that might affect the male to female ratio within each group?

Make predictions and test your ideas within this model. Feel free to manipulate more than one variable at a time.

As you are testing your hypotheses, you will notice that patterns are emerging from the data. For example, if you keep the number of people at the party constant but gradually increase the tolerance level, more mixed groups appear.

How high does the tolerance value have to be before you get mixed groups?

What percent tolerance tends to produce what percentage of mixing?

Thinking With Models

Using NetLogo to model situations like this party scenario allows you to experiment with a system in a rapid and flexible way that would be difficult to do in a real world situation. Modeling also gives you the opportunity to observe a situation or circumstance with less prejudice −− as you can examine the underlying dynamics of a situation. You may find that as you model more and more, many of your preconceived ideas about various phenomena will be challenged. For example, a surprising result of the Party model is that even a little intolerance tends to produce a great deal of separation between the sexes.

This is a classic example of an "emergent" phenomenon, where a group pattern results from the interaction of many individuals. This idea of "emergent" phenomena can be applied to almost any subject.

What other emergent phenomena can you think of?

To see more examples and gain a deeper understanding of this concept and how NetLogo helps learners explore it, you may wish to explore NetLogo's Models Library. It contains models that demonstrate these ideas in systems of all kinds.

For a longer discussion of emergence and how NetLogo helps learners explore it, see "Modeling Nature's Emergent Patterns with Multi−agent Languages" (Wilensky, 2001).

What's Next?

The section of the User Manual called Tutorial #1: Running Models goes into more detail about how to use the other models in the Models Library.

If you want to learn how to explore the models at a deeper level, Tutorial #2: Commands will introduce you to the NetLogo modeling language.

Eventually, you'll be ready for Tutorial #3: Procedures, where you can learn how to alter and extend existing models to give them new behaviors, and build your own models.

Tutorial #1: Models

If you read the Sample Model: Party section, you got a brief introduction to what it's like to interact with a NetLogo model. This section will go into more depth about the features that are available while you're exploring the models in the Models Library.

Sample Model: Wolf Sheep Predation

We'll open one of the Sample Models and explore it in detail. Let's try a biology model: Wolf Sheep Predation, a predator−prey population model.

- Open the Models Library from the File menu.

- Choose "Wolf Sheep Predation" from the Biology section and press "Open".

The Interface tab will fill up with lots of buttons, switches, sliders and monitors. These interface elements allow you to interact with the model. Buttons are blue; they set up, start, and stop the model. Sliders and switches are green; they alter model settings. Monitors and plots are beige; they display data.

When you first open the model, you will notice that the Graphics Window is empty (all black). To begin the model, you will first need to set it up.

- Press the "setup" button.

What do you see appear in the Graphics Window?

- Wait until the "setup" button pops back up (in other words, turns blue again).
- Press the "go" button to start the simulation.

As the model is running, what is happening to the wolf and sheep populations?

- Press the "go" button to stop the model.

Controlling the Model: Buttons

When a button is pressed, the model responds with an action. A button can be a "once" button, or a "forever" button. You can tell the difference between these two types of buttons by a symbol on the face of the button. Forever buttons have two arrows in the bottom right corners, like this:

Once buttons don't have the arrows.

Once buttons do one action and then stop. When the action is finished, the button pops back up. (Note: If you do not wait for the button to pop back up before pressing any other buttons, the model might get confused, and you may receive get an error message.)

Most models, including Wolf Sheep Predation, have a once button called "setup" and a forever button called "go". Many models also have a once button called "go once" or "step once" which is like "go" except that it advances the model by one time step instead of continuously. Using this button lets you watch the progress of the model more closely.

Forever buttons do an action over and over again. When you want the action to stop, press the button again. It will finish the current action, then pop back up.

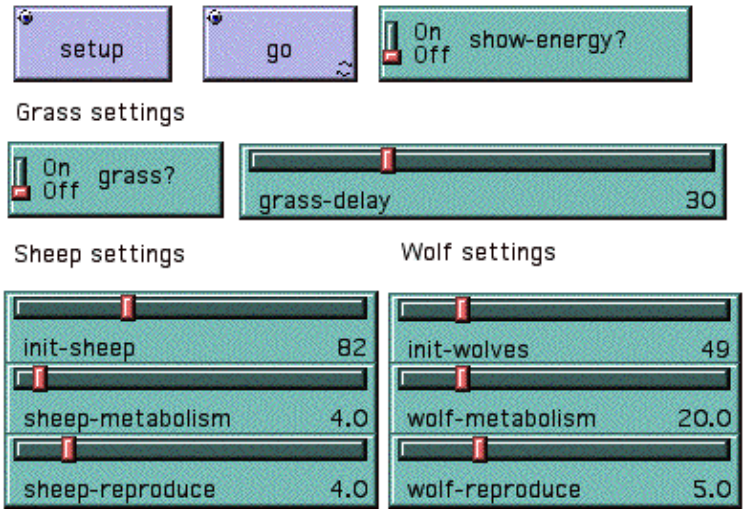
Stopping a forever button is the normal way to stop a model. It's safe to pause a model by stopping a forever button, then make it go on by pressing the button again. You can also stop a model with the "Halt" button on the Interface toolbar, but you should only do this if the model is stuck for some reason. The "Halt" button may interrupt the model in the middle of an action, and as the result the model could get confused.

- If you like, experiment with the "setup" and "go" buttons in the Wolf Sheep Predation model.

Adjusting Settings: Sliders and Switches

The settings within a model give you an opportunity to work out different scenarios or hypotheses. Altering the settings and then running the model to see how it reacts to those changes can give you a deeper understanding of the phenomena being modeled. Switches and sliders give you access to a model's settings.

Here are the switches and sliders in Wolf Sheep Predation:



Let's experiment with their effect on the behavior of the model.

- Open Wolf Sheep Predation if it's not open already.
- Press "setup" and "go" and let the model run for about a 100 ticks. (Note: there is a readout of the number of ticks right above the plot.)
- Stop the model by pressing the "go" button.

What happened to the sheep over time?

Let's take a look and see what would happen to the sheep if we change a few settings.

- Turn the "show−energy?" switch on.
- Press "setup" and let the model run for about the same amount of time as before.

What did this switch do to the model? Was the outcome the same as your previous run?

- Turn the "show−energy?" switch off.
- Using the same procedure as above, try out the "grass?" switch.

How does this switch affect what happens in the model?

The switches in Wolf Sheep Predation demonstrate that sometimes a switch or slider only changes how the model looks, while sometimes the switch or slider actually changes the behavior of model. For example, the "show−energy?" switch only changes what information is displayed, but the "grass?" switch actually makes the model behave differently.

Another type of setting is called a slider.

Sliders are a different type of setting then a switch. A switch has two values: on or off. A slider has a range of numeric values that can be adjusted. For example, the "initial−sheep" slider has a

minimum value of 0 and a maximum value of 250. The model could run with 0 sheep or it could run with 250 sheep, or anywhere in between. Try this out and see what happens. As you move the marker from the minimum to the maximum value, the number on the left side of the slider changes; this is the number the slider is currently set to.

Let's investigate Wolf Sheep Predation's sliders.

- Read the contents of the Information tab, located above the toolbar, to learn what each of this models' sliders represents.

The Information tab is extremely helpful in gaining insight into the model. Within this tab you will find an explanation of the model, suggestions on things to try, and other information. You may want to read the Information tab before running a model, or you might want to just start experimenting, then look at the Information tab later.

What would happen to the sheep population if there was more initial sheep and less initial wolves at the beginning of the simulation?

- Turn both of the switches off.
- Set the "init−sheep" slider to 100.
- Set the "init−wolves" slider to 20.
- Press "setup" and then "go".
- Let the model run for about 100 ticks.

What happened to the sheep population?

Did this outcome surprise you? What other sliders or switches can be adjusted to help out the sheep population?

- Set "sheep−reproduce" to 10.0
- Press "setup" and then "go".
- Let the model run for about 100 ticks.

What happened to the wolves in this run?

When you open a model, all the sliders and switches are on a default setting. If you open a new model or exit the program, your changed settings will not be saved, unless you choose to save them.

Gathering Information: Plots and Monitors

The purpose to modeling is to gather data on a subject or topic that would be very difficult to do in a laboratory situation. NetLogo has two main ways of displaying data to the user: plots and monitors.

Plots

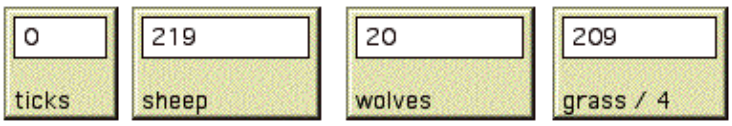
The plot in Wolf Sheep Predation contains three lines: wolf, sheep and grass. The lines show what's happening in the model over time. To see which line is which, click on "Pens" in the upper right

corner of the plot window to open the plot pens legend. A key appears that indicates what each line is plotting. In this case, it's the population counts. (Note: the grass count is divided by four so it doesn't make the graph too tall.)

If you want to save the data from a plot to view or analyze it in another program, you can use the "Export Plot" item on the File menu. It saves this information to your computer in a format that can by read back by spreadsheet and database programs such as Excel.

Monitors

Monitors are another method of displaying information in a model. Here are the monitors in Wolf Sheep Predation:



The monitor labeled "ticks" tells us how much time has passed in the model. The other monitors show us the population of sheep and wolves, and the amount of grass. (Remember, the amount of grass is divided by four to keep the graph from getting too tall.)

The numbers displayed in the monitors update continuously as the model runs, whereas the plots show you data from the whole course of the model run.

Note that NetLogo has also another kind of monitor, called "agent monitors". These will be introduced in Tutorial #2.

Changing Graphics Window Settings

Changing the settings in the Graphics Window allows you to make the NetLogo "world" bigger or smaller.

The size of the Graphics Window is determined by three separate settings: Screen Edge X, Screen Edge Y, and Patch Size. Let's take a look at what happens when we change the size of the Graphics Window in the "Wolf Sheep Predation" model.

- Place your mouse pointer next to, but still outside of, the Graphics Window.

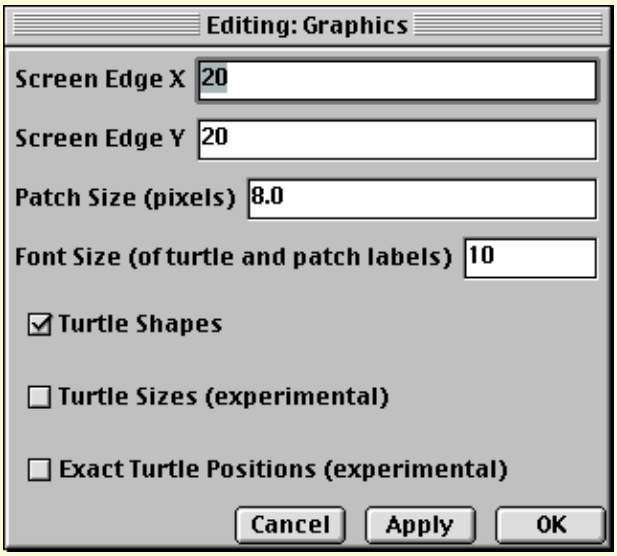
You will notice that the pointer turns into a crosshair.

- Hold down the mouse button and drag the crosshair over the Graphics Window.

The Graphics Window is now selected, which you know because it is now surrounded by a gray border.

- Press the Edit button on the Interface toolbar, near the top of the NetLogo window.

A dialog box will open containing all the settings for the Graphics Window:



What are the current settings for Screen Edge X, Screen Edge Y, and Patch Size?

- Press "cancel" to make this window go away.
- Select the Graphics Window again.
- Drag one of the square black "handles" at the edges and corners of the Graphics Window.
- Edit the Graphics Window again and look at the settings.

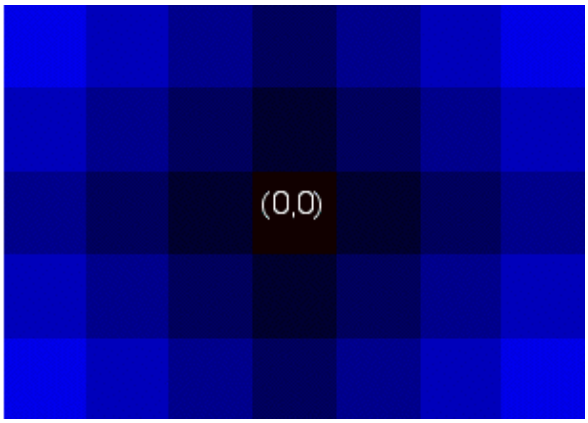
What numbers changed?

What numbers didn't change?

The NetLogo world is a two dimensional grid of "patches". Patches are the individual squares in the grid.

In Wolf Sheep Predation, when the "grass?" switch is on the individual patches are easily seen, because some of them are green, while others are brown.

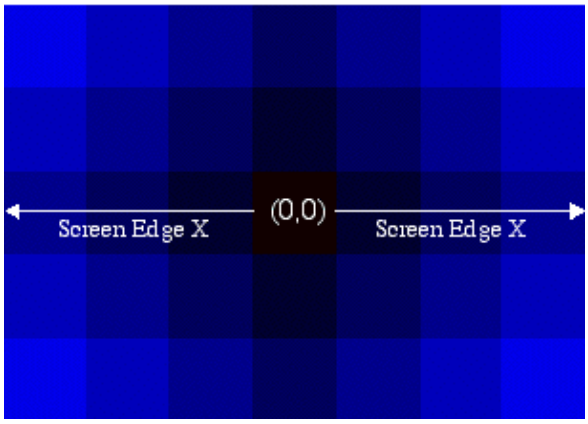
Think of the patches as being like square tiles in a room with a tile floor. Exactly in the middle of the room is a tile labeled (0,0); meaning that if the room was divided in half one way and then the other way, these two dividing lines would intersect on this tile. We now have a coordinate system that will help us locate objects within the room:

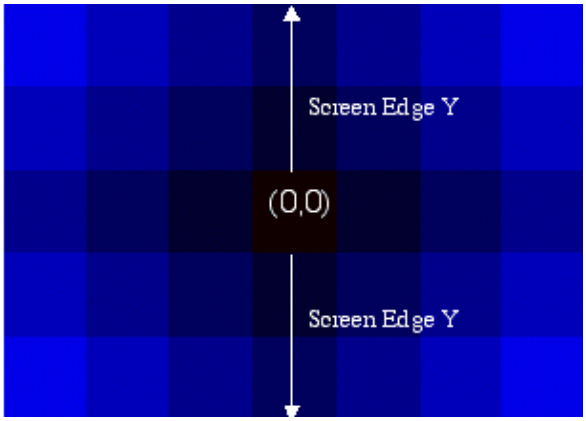


How many tiles away is the (0,0) tile from the right side of the room?

How many tiles away is the (0,0) tile from the left side of the room?

In NetLogo, the distance the middle tile is from the right or left edge of the room this is called Screen Edge X. And the distance the middle tile is from the top and bottom edges is called Screen Edge Y:





In these diagrams, Screen Edge X is 3 and Screen Edge Y is 2.

When you change the patch size, the number of patches (tiles) doesn't change, the patches only get larger or smaller on the screen.

Let's look at the effect of changing Screen Edge X and Screen Edge Y.

- Using the Edit dialog that is still open, change Screen Edge X to 30 and Screen Edge Y value to 10.

What happened to the shape of the Graphics Window?

- Press the "setup" button.

Now you can see the new patches you have created.

- Edit the Graphics Window again.
- Change the patch size to 20 and press "OK".

What happened to the size of the Graphics Window? Did its shape change?

Editing the Graphics window also lets you change two other settings: the font size of labels, and whether the Graphics Window uses shapes. Feel free to experiment with these settings as well. In order to see the effect of the font size settings, you'll need to have the "show−energy?" switch turned on.

Once you are done exploring the Wolf Sheep Predation model, you may want to take some time just to explore some of the other models available in the Models Library.

The Models Library

The library contains four sections: Sample Models, Code Examples, HubNet, Unverified Models.

Sample Models

The Sample Models section is organized by subject area and currently contains more than 60 models. We are continuously working on adding new models to it, so come visit this section at a later date to view the new additions to the library.

Code Examples

These are simple demonstrations of particular features of NetLogo. They'll be useful to you later when you're extending existing models or building new ones. For example, if you wanted to put a histogram within your model, you'd look at "Histogram Example" to find out how.

HubNet Activities

This section contains participatory simulations for use in the classroom. For more information about HubNet, see the HubNet Guide.

Unverified Models

These models are still in the process of being tested and reviewed for content and accuracy. They are complete and functional; we just haven't tested them as thoroughly, or made sure the code is good sample code.

What's Next?

If you want to learn how to explore models at a deeper level, Tutorial #2: Commands will introduce you to the NetLogo modeling language.

In Tutorial #3: Procedures you can learn how to alter and extend existing models and build your own models.

Tutorial #2: Commands

In Tutorial #1, you had the opportunity to view some of the NetLogo models, and you have successfully navigated your way through the menu bar, the toolbars, opening and running models, and gathering information from a model. In this section, the focus will start to shift from observing models to manipulating models. You will start to see the inner workings of the models and be able to change how they look.

Sample Model: Traffic Basic

- Go to the Models Library (File menu).
- Open up Traffic Basic, found in the "Social Science" section.
- Run the model for a couple minutes to get a feel for it.
- Consult the Information tab for any questions you may have about this model.

In this model, you will notice one red car in a stream of blue cars. The stream of cars are all moving in the same direction. Every so often they "pile up" and stop moving. This is modeling how traffic jams can form without any cause such as an accident, a broken bridge, or an overturned truck. No "centralized cause" is needed for a traffic jam to form.

You may alter the settings and observe a few runs to get a full understanding of the model.

As you are using the Traffic Basic model, have you noticed any additions you would like to make to the model?

Looking at the Traffic Basic model, you may notice the environment is fairly simple; a black background with a white street and number of blue cars and one red car. Changes that could be made to the model include: changing the color and shape of the cars, adding a house or street light, creating a stop light, or even creating another lane of traffic. Some of these suggested changes are cosmetic and would enhance the look of the model while the others are more behavioral. We will be focusing more on the simpler or cosmetic changes throughout most of this tutorial. (Tutorial #3 will go into greater detail about behavioral changes, which require changing the Procedures tab.)

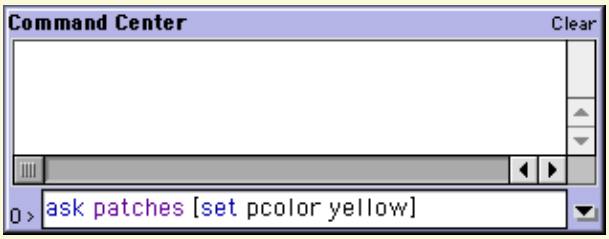
To make these simple changes we will be using the Command Center.

The Command Center

The Command Center is located in the Interface Tab and allows you to enter commands or directions to the model (without changing the actual mechanisms of the model). Commands are instructions you can give to NetLogo's agents: turtles, patches, and the observer.

In Traffic Basic:

- Press the "setup" button.
- Locate the Command Center.
- Click the mouse in the white box at the bottom of the Command Center.
- Type the text shown here:



- Press the return key.

(Refer to the Interface Guide for details explaining the different parts of the Command Center.)

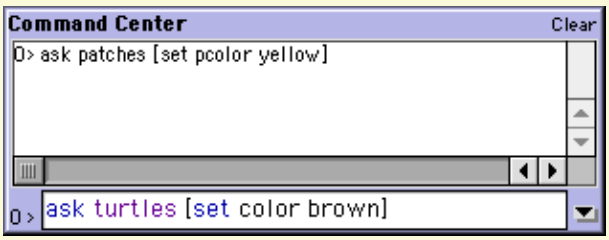
What happened to the Graphics Window?

You may have noticed the background of the Graphics Window has turned all yellow and the street has disappeared.

Why didn't the cars turn yellow too?

Looking back at the command that was written, we asked only the patches to change their color. In this model, the cars are represented by a different kind of agent, called "turtles". Therefore, the cars did not received these instructions and thus did not change.

- Type in the Command Center the text shown below:



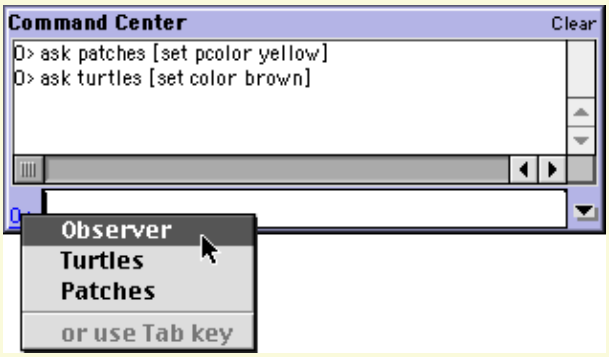
Was the result what you expected?

Your Graphics Window should have a yellow background with a line of brown cars in the middle of the window:

The NetLogo world is a two dimensional world that is made up of turtles, patches and an observer. The patches create the ground in which the turtles can move around on and the observer is a being that oversee everything that is going on in the world. (For a detailed description and specifics about this world, refer to the NetLogo Programming Guide.)

In the Command Center, we have the ability to give an observer a command, turtles a command, or patches a command. We choose these by using the popup menu located in the bottom left corner of the Command Center. (You can also use the tab key on your keyboard to choose between them.)

- In the Command Center, click on the "O>":



- Choose "Turtles" from the popup menu (or use the tab key).
- Type `set color pink` and press return.
- Choose "Patches" from the popup menu (or use the tab key).
- Type `set pcolor white` and press return.

What does the Graphics Window look like now?

What was omitted in these two commands but not in the observer command, from the earlier exercise?

The observer oversees the world and therefore can give a command to the patches or turtles using `ask`. Like in the case above, the observer has to ask the patches to set their color to yellow. But when a command is directly given to a group of agents, like in the second example, you only have to give the command itself.

- Press "setup".

What happened?

Why did the Graphic Window revert back to the old version, with the black background and white road? This is because the effects of commands given in the Command Center are temporary. Upon pressing the "setup" button, the model will reconfigure itself back to the settings outlined in the Procedures tab. The purpose of the Command Center is not to permanently change the model. It is a tool to customize current models and allows for you to manipulate the NetLogo world to further answer those "What if" questions that pop up as you are investigating the models. (The Procedures tab is explained in the next tutorial, and in the Programming Guide.)

Now that we have familiarized ourselves with the Command Center, let's work on our list of changes we wanted to make to Traffic Basic.

Working With Colors

You may have noticed in the previous section that we used two different words for changing color: `color` and `pcolor`.

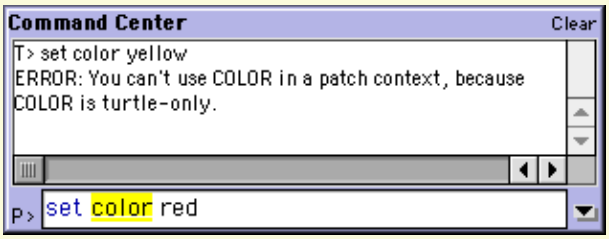
What is the difference between color and pcolor?

- Choose "Turtles" from the popup menu in the Command Center (or use the tab key).
- Type `set color blue` and press return.

What happened to the cars?

Think about what you did to make the cars turn blue, and try to make the patches turn red.

If you try to ask the patches to `set color red`, an error message occurs:



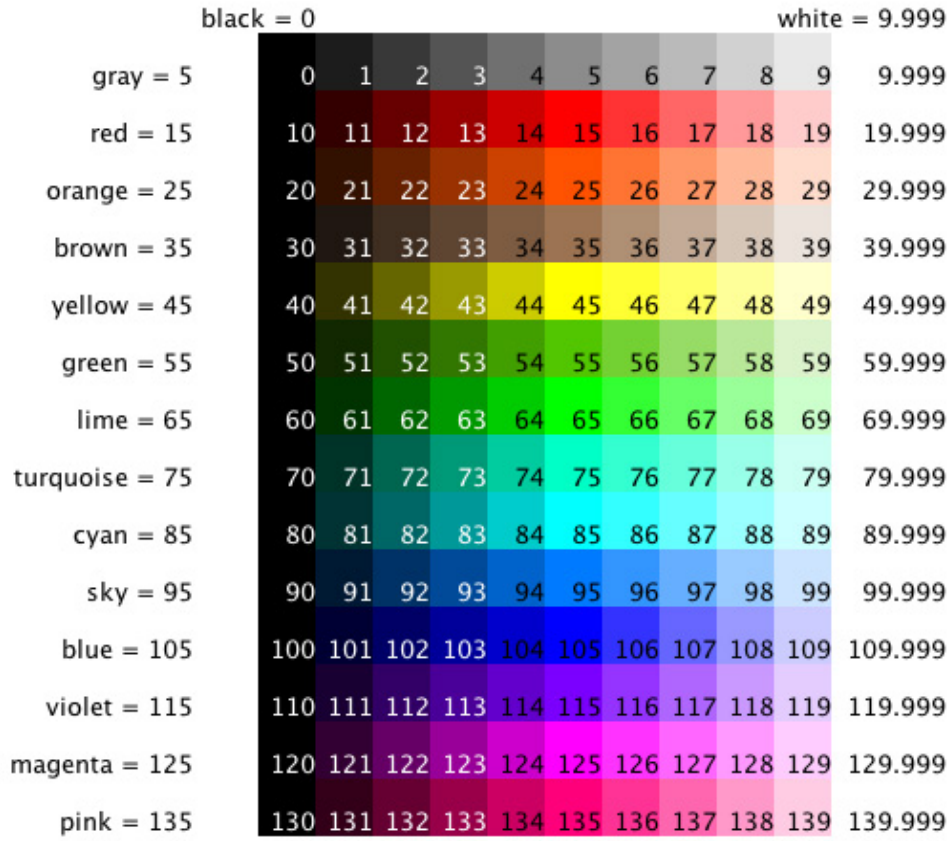
- Type `set pcolor red` instead and press return.

We call `color` and `pcolor` "variables". Some commands and variables are specific to turtles and some are specific to patches. For example, the `color` variable is a turtle variable, while the `pcolor` variable is a patch variable.

Go ahead and practice altering the colors of the turtles and patches using the `set` command and these two variables.

To be able to change patches' and turtles' colors, or shall we say cars and backgrounds, we have to gain a little insight into how NetLogo deals with colors.

In NetLogo, all colors have a numeric value. In all of the exercises we have been using the name of the color. This is because NetLogo recognizes 16 different color names. This does not mean that NetLogo only recognizes 16 colors. There are many shades in between these colors that can be used too. Here's a chart that shows the whole NetLogo color space:



To get a color that doesn't have its own name, you just refer to it by a number instead, or by adding or subtracting a number from a name. For example, when you type `set color red`, this does the same thing as if you had typed `set color 15`. And you can get a lighter or darker version of the same color by using a number that is a little larger or a little smaller, as follows.

- Choose "Patches" from the popup menu in the Command Center (or use the tab key).
- Type `set pcolor red - 2`

By subtracting from red, you make it darker.

- Type `set pcolor red + 2`

By adding to red, you make it lighter.

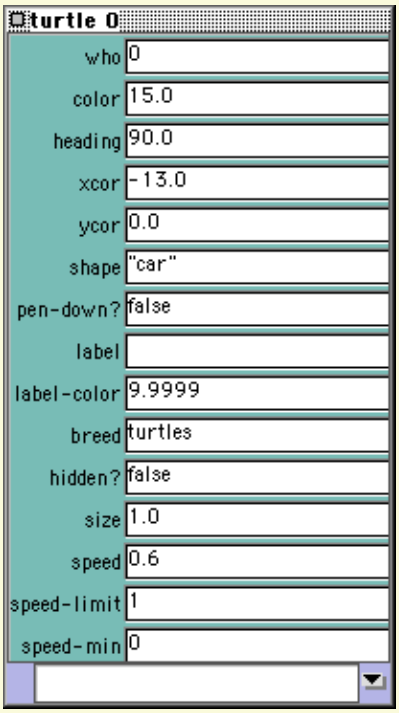
You can use this technique on any of the colors listed in the chart.

Agent Monitors and Agent Commanders

In the previous activity, we used the set command to change the colors of all the cars. But if you recall, the original model contained one red car amongst a group of blue cars. Let's look at how to change only one car's color.

- Press "setup" to get the red car to reappear.
- If you are on a Macintosh, hold down the Control key and click on the red car. On other operating systems, click on the red car with the right mouse button.
- From the popup menu that appears, choose "inspect turtle 0"

A turtle monitor for that car will appear:



Taking a closer look at this turtle monitor, we can see all of the variables that belong to the red car. A variable is a place that holds a value that can be changed. Remember when it was mentioned that all colors are represented in the computer as numbers? The same is true for the agents.

Let's take a closer look at the turtle monitor:

What is this turtle's ID number?

What color is this turtle?

What shape is this turtle?

This turtle monitor is showing a turtle who that has an ID number of 0, a color of 15 (red −− see above chart), and the shape of a car.

There are three ways to change an individual turtle's color.

One way is to use the box found at the bottom of the monitor, also called an Agent Commander. You type commands here, just like in the Command Center, but the commands you type here are only done by this particular turtle.

- Type `set color pink`

What happens in the Graphics Window?

Did anything change in the Turtle Monitor?

A second way to change one turtle's color is to go directly to the color variable in the Turtle Monitor and change the value.

- Select the text to the right of "color" in the Turtle Monitor.
- Type in a new color such as `green + 2`.

What happened?

There are two other ways to open a turtle monitor besides the one you just did. One way is to choose "Turtle Monitor" from the Tools menu, then type the ID number of the turtle you want to inspect into the "who" field. The other way is to type `inspect turtle 0` (or other ID number) into the Command Center.

You close a turtle or patch monitor by clicking the close box in the upper left hand corner (Macintosh) or upper right hand corner (other operating systems).

Just as there are Turtle Monitors, there are also Patch Monitors.

Can you make a patch monitor and use it to change the color of a single patch?

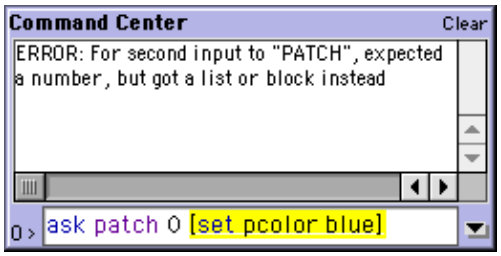
The third way to change an individual turtle's or patch's color is to use the observer. Since, the observer oversees the NetLogo world, it can give commands that affect individual turtles or patches, as well as groups of turtles or patches.

- In the Command Center, select "Observer" from the popup menu (or use the tab key).
- Type `ask turtle 0 [set color blue]` and press return.

What happens?

Can you turn one patch blue?

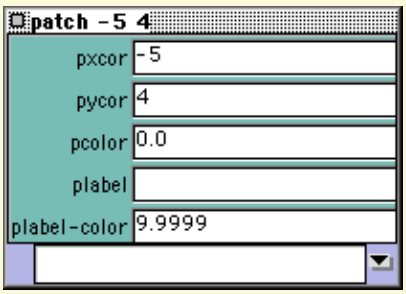
If you try to have the observer `ask patch 0 [set pcolor blue]`, you'll get an error message:



To ask an individual turtle to do something, we use its ID number. But patches don't have ID numbers, therefore we need to refer to them some other way.

Remember, patches are arranged on a coordinate system. Two numbers are needed to plot a point on a graph: an x−axis value and a y−axis value. Patch locations are designated in the same way as plotting a point.

- Open a patch monitor for any patch.



The monitor shows that for the patch in the picture, its `pxcor` variable is −5 and its `pycor` variable is 4. If we go back to the analogy of the coordinate plane and wanted to plot this point, the point would be found in the upper left quadrant of the coordinate plane where x=−5 and y=4.

To tell this particular patch to change color, use its coordinates.

- Type `ask patch −5 4 [set pcolor blue]` and press return.

What are the two words in this command that "tip you off" that we are addressing a patch?

What's Next?

At this point, you may want to take some time to try out the techniques you've learned on some of the other models in the Models Library.

In Tutorial #3: Procedures you can learn how to alter and extend existing models and build your own models.

Tutorial #3: Procedures

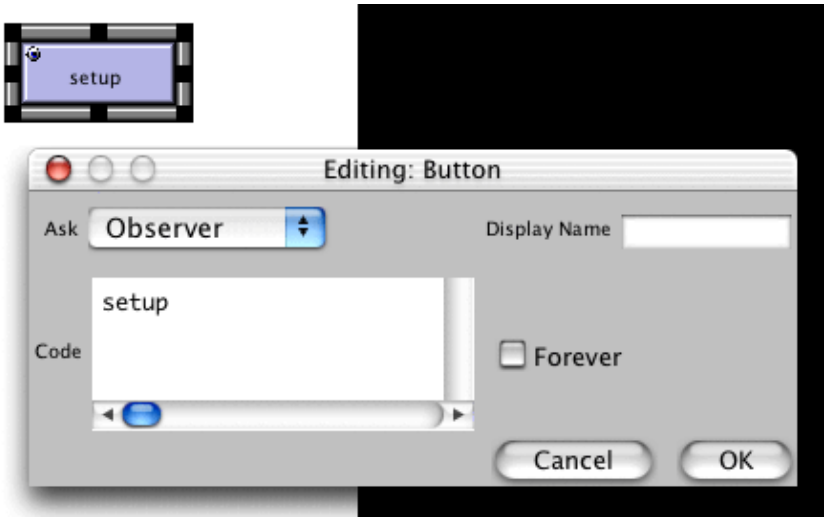
In Tutorial #2, you learned how to use agent monitors and command centers to inspect and modify agents and make them do things. Now you're ready to learn about the real heart of a NetLogo Model: the Procedures tab. This tutorial leads you through the process of building a complete model, built up stage by stage, with every step explained along the way.

Setup and Go

To start a new model, select "New" from the the File menu. Then begin making your model by creating a once−button called 'setup'.

Here's how to make the button:

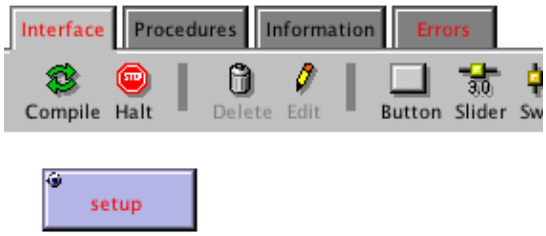
1. click on the button icon in the Toolbar
2. click where you want the button to be in the empty white area of the Interface tab
3. when the dialog box for editing the properties of the button opens, type `setup` in the box labeled "Code"



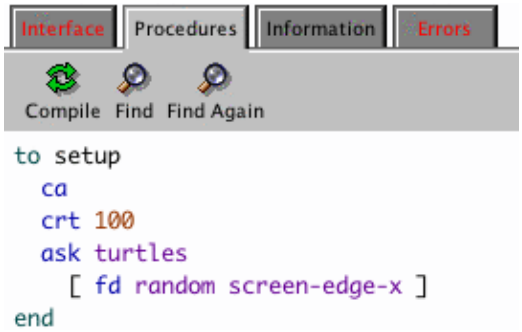
4. press "OK" to dismiss the dialog box

Now you have a button called 'setup'. It will execute the procedure 'setup' when pressed, which once we define it, will do just that −− set up the NetLogo world the way we want it.

At this point, the Errors tab will turn red. That's because there is no procedure called 'setup'! If you want to see the actual error message, switch to the Errors tab:



Now switch to the Procedures Tab and create the 'setup' procedure like this:



One line at a time:

to setup begins defining a procedure named "setup".

ca is short for **clear−all** (you can also spell it out if you want). This command will blank out the screen, initialize any variables you might have to 0, and kill all turtles. Basically, it wipes the slate clean for a new run of the project.

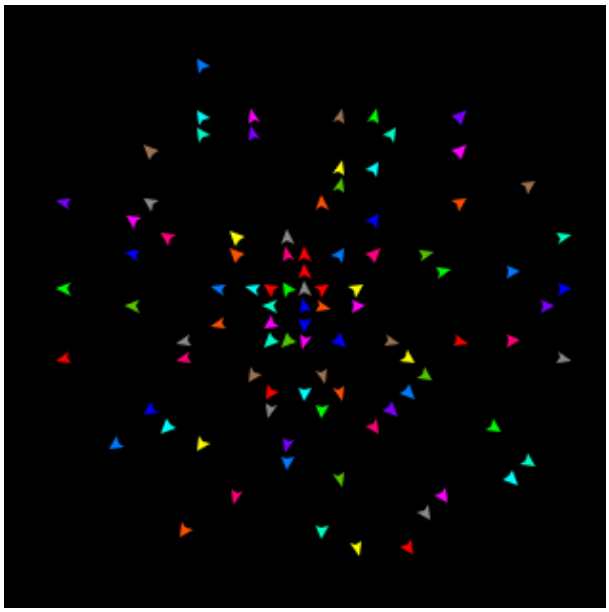
crt 100 will then create 100 turtles. Each of these turtles will begin on the center patch (at location 0,0). That's why you only see what looks like one turtle on the screen; they're all on top of each other −− lots of turtles can share the same patch. Only the last turtle to arrive on the patch is visible. Each of these newly−created turtles has its own color, evenly distributed through the range of NetLogo colors, and its own heading, evenly distributed around the circle.

ask turtles [...] tells each turtle to execute, independently, the instructions inside the brackets. Note that **crt** is not in brackets. If the agent is not specified, the observer executes it.

fd (random screen−edge−x) is really a pair of commands. Each turtle will first evaluate the expression **random screen−edge−x** which will return a random value between 0 and 'screen−edge−x' (the dimension from the center to the edge of the screen along the x−axis, maybe 20 or so). It then takes this number, and goes **fd** (short for **forward**) that number of step, in the direction of its heading.

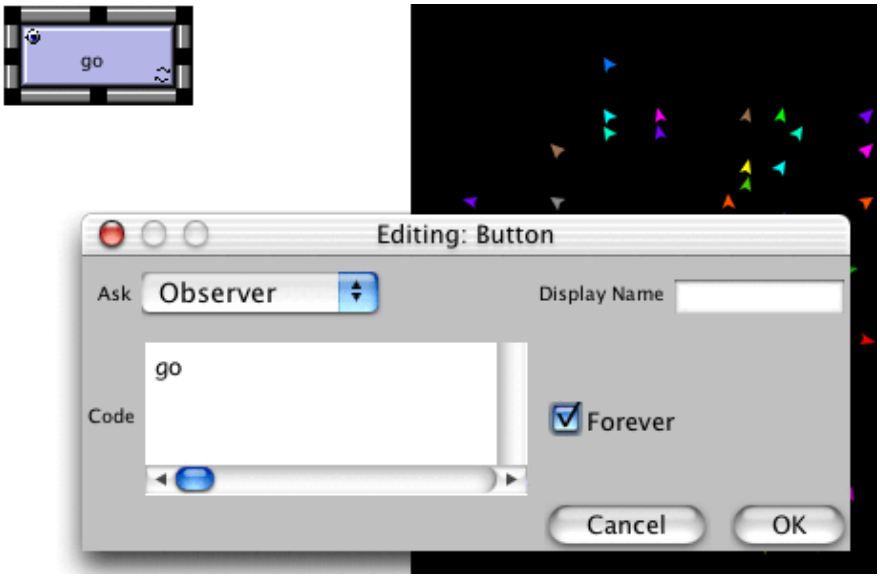
end ends the definition of the "setup" procedure.

Press your 'setup' button when you've written the code. You will see the turtles quickly spread out in a rough cluster:



Notice the density distribution of the turtles on the Graphics Window. Press 'setup' a couple more times, and watch how the turtles' arrangement changes. Can you think of other ways to randomly distribute the turtles over the screen? (There are lots of ways.) Note that if a turtle moves off the screen, it "wraps", that is, comes in the other side. When you're satisfied that the button does what you want, it's time to write your next procedure.

Make a forever−button called 'go'. Again, begin by creating a button, but this time check the "forever" checkbox in the edit dialog.



Then write its procedure:

```
to go
  move-turtles
end
```

But what is ***move−turtles***? Is it a primitive (built in to NetLogo), like **fd** is? No, it's a procedure that you're about to write, right after the **go** procedure:

```
to move−turtles
  ask turtles [
    rt random 360
    fd 1
  ]
end
```

Line by line:

ask turtles [ ] says that each turtle should execute the commands in the brackets.

rt random 360 is another two−step command. First, each turtle picks a random integer between 0 and 359 (**random** doesn't include the number you give it). Then it turns right ("rt" is short for "right turn") that far. Heading is measured in degrees, clockwise around the circle, starting with 0 degrees at twelve o'clock.

fd 1: After each turtle does that, it moves forward one step in this new direction.

Why couldn't we have just written that in ***go***? We could, but during the course of building your project, it's likely that you'll add many other parts. We'd like to keep ***go*** as simple as possible, so that it is easy to understand. It could include many other things you want to have happen as the model runs, such as calculating something or plotting the results. Each of these sub−procedures could have its own name.

The 'go' button is a forever−button, meaning that it will continually execute its code until you shut it off (by clicking on it again). After you have pressed 'setup' once, to create the turtles, press the 'go' button. Watch what happens. Turn it off, and you'll see that all turtles stop in their tracks.

We suggest you start experimenting with other turtle commands. Type commands into the Command Window (like **set color** *red*), or add them to ***setup***, ***go***, or ***move−turtles***. Note that when you enter commands in the Command Center, you must choose **T>**, **P>**, or **O>** in the popup menu on the left, depending on which agents are going to execute the commands. You can also use the tab key, which is more convenient than using the popup menu. **T>*commands*** is identical to **O> ask turtles [*commands*]**, and **P>*commands*** is identical to **O> ask patches [*commands*]**.

You might try typing **T> pendown** into the Command Center, or changing **set heading** (**random 360**) to **lt** (**random 45**). Or, instead of saying **crt** *100* in ***setup***, say **crt** *number*, where *number* is a slider variable, and create a slider from the toolbar just as you created the buttons. Play around. It's easy and the results are immediate and visible −− one of NetLogo's many strengths. Regardless, the tutorial project continues...

Patches and Variables

Now we've got 100 turtles aimlessly moving around, completely unaware of anything else around them. Let's make things a little more interesting by giving these turtles a nice background against which to move. Go back to the 'setup' procedure. We can rewrite it as follows:

```
patches−own [elevation]
```

```
to setup
  ca
  setup-patches
  setup-turtles
end

to setup-patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]
end

to setup-turtles
  crt 100
  ask turtles
    [ fd (random screen-edge-x) ]
end
```

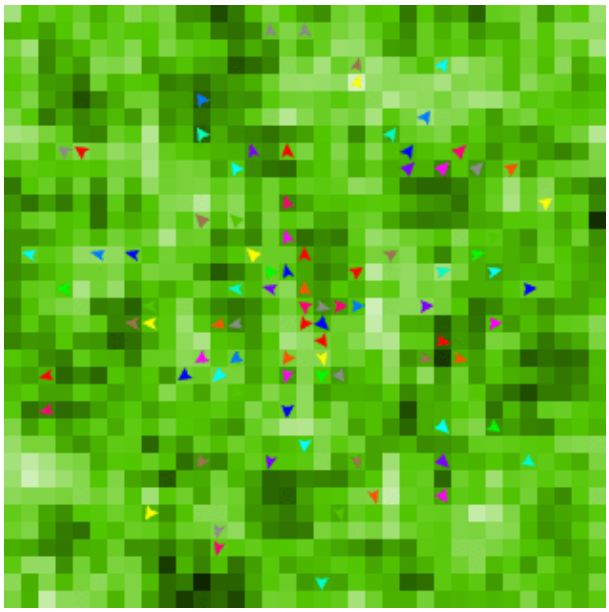
The line at the top, **patches-own [elevation]** declares that we have a variable for the patches, called ***elevation*.** Our **setup-patches** procedure then uses this variable. First, each patch picks a random number between 0 and 9999, inclusive, and sets its *elevation* value to that number.

We then use an observer primitive, **diffuse**, that essentially smooths out the distribution of this variable over the neighboring patches.

We can see the effect of **diffuse** in the next command, which uses **scale-color**, a reporter uses the different values of *elevation* to assign colors to the patches. In this case, we're assigning different shades of green to all the patches. (Don't worry about the numbers given to **diffuse** and **scale-color** just yet...) The larger *elevation* is, the lighter the shade of green. Low values of *elevation* will result in darker shades.

Setup-turtles is familiar, being exactly what we were doing in our old **setup**.

Type this in and press the 'setup' button. Voila --a lush NetLogo patch landscape.



Here's how you can see how **diffuse** works. Return to the Procedures Window, and 'comment−out' the diffuse command, by adding a semicolon in front of it like this:

```
;diffuse elevation 1
```

Press 'setup' again −− doesn't look as good, does it? This is because, as mentioned above, **diffuse** has each patch share its value of *elevation* with all its neighbors, by having every patch reset its value of *elevation* to a new value that depends on the value of *elevation* all around it. Don't worry if you don't exactly understand what's going on here. You can read about it by going to the Primitives Dictionary; and it may help to toy with the values being passed to it, and see what happens.

We're now prepared to create some kind of dialog between the turtles and the patches. In fact, we even have an idea for a project here. Notice that we called the patch variable 'elevation', and that our landscape sort of looks topographical? We're going to have our turtles do what is called 'hill−climbing', where every turtle seeks to find the highest elevation it can.

We'll now explain the notion of compound commands −− a feature that makes NetLogo quite different from StarLogoT (and somewhat closer to natural language). Go to the Command Center, and type **O> show max values−from patches [*elevation*]** and **show min values−from patches [*elevation*]**. These two reporters will, respectively, search over all the patches to return to you the highest elevation and the lowest. These commands work like this (you can read about them in the NetLogo Dictionary too):

Look up 'values−from' in the dictionary. It shows "values−from AGENTSET [expression]" and says it returns a list. In this case, it looks at the expression (elevation) for each agent in the agentset (patches) and returns all of these as a list of elevations.

Look up 'min' in the dictionary. It shows "min *list*" and says it's a reporter. So it takes the list of elevations and reports the smallest value.

'Show' displays this value in the command center. And there you have it. We will use this compound command in our model.

One last thing before we return to the coding. Those two values we just read, the highest and the lowest elevations, might be nice to know. Rather than have to retype that every time, let's use a shortcut. First, at the top of your code (right after the 'patches−own' declaration), declare two global variables as such:

```
globals [highest ;; the highest patch elevation
         lowest] ;; the lowest patch elevation
```

and in ***setup−patches***, write:

```
to setup−patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale−color green elevation 1000 9000 ]
  set highest max values−from patches [elevation]
  set lowest min values−from patches [elevation]
  ask patches [
    if (elevation > (highest − 100))
```

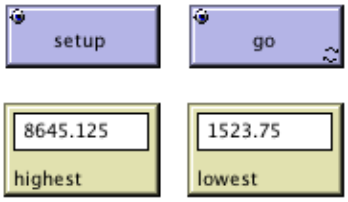
```
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black] ]
end
```

Now we have saved the highest and lowest points in our terrain and displayed them graphically.

Look at the last two statements, the '**if**−statements'. Each patch looks at both of these statements, and compares its own value of *elevation* to our global variables *highest* and *lowest*. If the comparison evaluates to 'true', the patch executes the list of commands inside the brackets. In this case, it simply changes its color. If the comparison evaluates to 'false', the patch skips over the commands.

What this does is that all patches whose value of *elevation* is NEAR to the highest (within about 1% for our values) change their color to white, and all patches whose values are NEAR to the lowest become black. This is so that they'll be easier to see. You can make a couple of quick changes here if you wish −− they won't affect the rest of the model. Instead of saying 'set pcolor white' and 'set pcolor black', you can say 'set pcolor blue' and 'set pcolor red', or whatever other colors you wish. Also, you can change the range of 'highest peaks' and 'lowest peaks' by changing the number 100 to some other number.

After this, create two monitors in the Interface Window with the Toolbar. (You make them just like buttons and sliders, using the monitor icon on the Toolbar.) Name one of them 'highest' and one of them 'lowest'. Now every time you click 'setup' and redistribute the values of *elevation*, you'll know exactly what the highest values are, and where they can be found.



An Uphill Algorithm

Okay. Finally we're ready to start hill−climbing. To rehash: we've got some turtles randomly spread out from the origin; and we've got a landscape of patches, whose primary attribute is their *elevation*. Lastly, we have two kinds of tools to help us understand the patch landscape: each patch has a color, depending on its value of *elevation*, and we have a pair of monitors telling us what the highest peak and lowest valley are. What we need now is for the turtles to wander around, each trying to get to the patch that has the highest elevation.

Let's try a simple (i.e. naive) algorithm first. We'll assume a three things: 1), that the turtles cannot see ahead farther than just one patch; 2), that each turtle can move only one square each turn; and 3), that turtles are blissfully ignorant of each other. Before, we had a procedure ***move−turtles*** like this:

```
to move−turtles
  ask turtles [
    set heading (random 360)
    fd 1
  ]
```

```
end
```

But now we don't want them to move randomly about. We want each turtle to look at the *elevation* of each patch around it, and move to the patch with the highest elevation (i.e., execute a greedy search, if you will). If none of the patches around it have a higher elevation than the patch it is on, it'll stay put. Here's the code:

```
;; performs a greedy-search of radius 1
;; for highest elevation
to move-to-local-max   ;; call this procedure from 'go'
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
    [ fd 1 ]
  ]
end
```

There are five new commands here: '***uphill***', '***elevation*−of**', '***patch−at***', and '***dx***' and '***dy***'. 'uphill elevation' finds the heading to the patch with the highest value of *elevation* in the patches in a one−patch radius of the turtle. Then through the use of the command 'set heading', the turtle sets its heading to that direction. 'elevation−of patch−at dx dy' has each turtle looks at the variable *elevation* in the patch on which the turtle would be if it went forward 1. If the test is true, the turtle moves itself forward 1. (The test is necessary because if the turtle is already on the peak, we don't want it to move off it!)

Go ahead and type that in, but before you test it out by pressing the 'go' button, ask yourself this question: what do you think will happen? Try and predict how a turtle will move, where it will go, and how long it'll take to get there. When you're all set, press the button and see for yourself.

Not too exciting. Surprised? Try to understand why the turtles converge to their peaks so quickly. Maybe you don't believe the algorithm we've chosen 'works correctly'. There's a simple change you can make to test it: rewrite ***setup−patches*** so that it says:

```
to setup-patches
  ask patches
  [
    set elevation pycor
    set pcolor scale-color green elevation
                (0 - screen-edge-y) screen-edge-y
  ]
end
```

Now run the project. See that the turtles all head for the highest elevation −− the top of the screen.

Another common tool to see what's going on is to write T> pd in the Command Center. Then each turtle traces its path with its color, and you can observe its path.

Our turtles rapidly arrive at local maxima in our landscape. Local maxima and minima abound in a randomly generated landscape like this one. Our goal is to still get the turtles to find an 'optimal maximum', one of the white patches. We need to start refining the model.

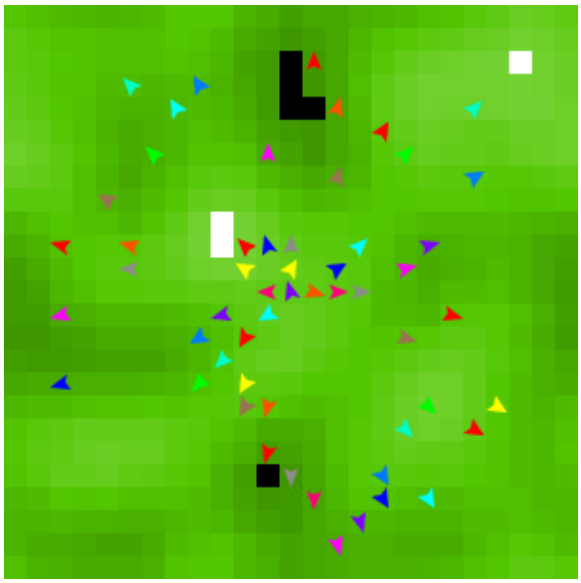
Part of the problem is that our terrain is terribly lumpy. Every patch picked a random elevation, and then we diffused these values one time. This really doesn't give us a continuous spread of elevation across the graphics window, as you might have noticed. We can correct this problem to an arbitrary

degree by diffusing more often. Replace the line:

```
diffuse elevation 1
```

with:

```
repeat 5 [ diffuse elevation 1 ]
```



The **repeat** primitive is another way for NetLogo to loop besides the forever−button. **Repeat** takes a number (here, 5) and a command list (here, the **diffuse** command), and executes the command list that number of times (here, five times). Try it out, and look at the landscape (i.e. press 'setup' and see what you think). Then, press 'go' and watch the turtles' performance. (Remember that the lighter the patch, the greater the elevation.)

Obviously, fewer peaks make for an improvement in the turtles' performance. On the other hand, maybe you feel like this is cheating −− the turtles really aren't doing any better, it's just that their problem was made easier. True enough. If you call **repeat** with an even higher number (40 or so), you'll end up with only a handful of peaks, as the values become more evenly distributed with every successive call. Watch your monitor values.

In order to specify how 'smooth' you want your world to be, let's make it easier to try different values. Maybe one time you'll want the turtles to try and 'solve a hard world', and maybe another time you'll just want to look at an easy landscape. So we'll make a slider variable. Create a slider in the Interface Window and call it "smoothness" in the editing box. The minimum can be 0, and the maximum can be 50 or so. Then change your code to:

```
repeat smoothness [ diffuse elevation 1 ]
```

Experiment with the turtles' performance in different worlds.

We still haven't even begun to solve the greedy−search problem, though. Before trying something else, it'd be nice if we could have some other, more precise method for evaluating the turtles' performance; watching little arrows writhe about in the Graphics Window only helps so much. Fortunately, NetLogo allows us to plot data as we go along.

To make plotting work, we'll need to create a plot in the Interface tab, and set some settings in it. Then we'll add one more procedure to the Procedures tab, which will update the plot for us.

Let's do the Procedures tab part first. Change **go** to call the new procedure we're about to add:

```
to go
  move-turtles
  do-plots
end
```

Then add the new procedure:

```
to do-plots
   set-current-plot "Turtles at Peaks"
   plot count turtles with
     [ elevation >= (highest - 100) ]
end
```

What we're plotting is the number of turtles who've reached our 'peak-zone' (within 1% of the highest elevation) at some given time.

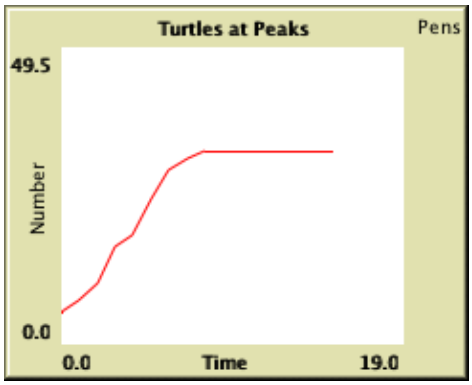
Note that we use the **plot** primitive to add the next point to a plot, but before doing that, we need to tell NetLogo which plot we want, since later our model might have more than one plot.

Thus we're plotting the number of turtles within 100 units of our maximum elevation at some given point in time. The **plot** command moves the current plot pen to the point that has x- coordinate equal to 1 greater than the old x- coordinate and y-coordinate equal to the value given in the plot command (in this case, the number of turtles with [elevation >= (highest - 100)]). Then the *plot* command draws a line from the current position of the plot pen to the last point it was on.

In order for `set-current-plot "Turtles at Peaks"` to work, you'll have to add a plot to your model in the Interface tab, then edit it so its name is "Turtles at Peaks", the exact same name used in the code. Even one extra space will throw it off -- it really must be exactly the same in both places.

Note that when you create the plot you can set the minimum and maximum values on the x and y axes, and the color of the default plot pen (pick any color you like). You'll want to leave the "Autoplot?" checkbox checked, so that if anything you plot exceeds the minimum and maximum values for the axes, the axes will automatically grow so you can see the whole plot.

Now reset the project and run it again. You can now watch the plot be created as the model is running:



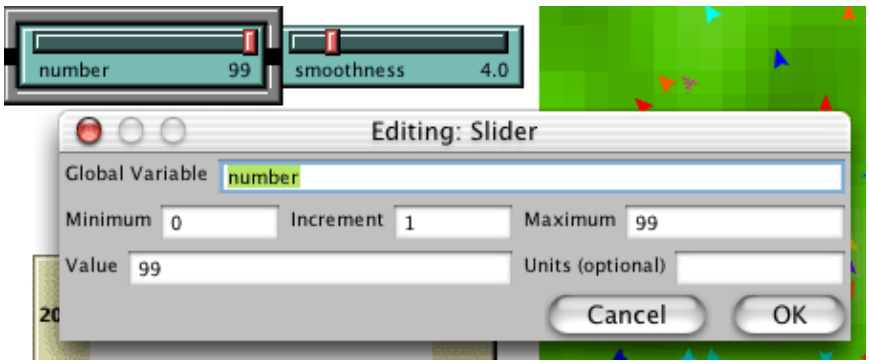
You might try running the model several times under different settings (i.e. different values of *smoothness*) and watch how fast the plot converges to some value, and what fraction of the turtles make it to the top.

Some More Details

There are a few miscellaneous bugs and quirks you might have already noticed. Here are some quick changes you can make: First, we have a green landscape −− a naturally green turtle is going to be hard to see. In the ask turtles block in 'setup−turtles', you can say:

```
if (color = green)
  [ set color red ]
```

Second, instead of always using 100 turtles, you can have a variable number of turtles. Make a slider variable (say, 'number'):



Then instead of 'crt 100', you can type:

```
crt number
```

How does using more or fewer turtles affect the success value displayed by the plot?

Third, when all the turtles have found their local maxima, wouldn't it be nice for the model to stop? This requires a few lines of code.

- Add a global variable *turtles−moved?* to the "globals" list:

```
globals [
  highest          ;; maximum patch elevation
```

```
  lowest          ;; minimum patch elevation
  turtles-moved?  ;; so we know when to stop the model
]
```

- At the end of the **go** procedure, add a test to see if any turtles have moved.

```
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
  [ stop ]
end
```

- In **move-to-local-max** if a turtle moves, set *turtles-moved?* to true.

```
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
    [
      fd 1
      set turtles-moved? true
    ]
  ]
end
```

Finally, what rules can you think of that would help turtles escape from lower peaks and all get to the highest ones? Try writing them.

What's Next?

So now you have a nice framework for exploring this problem of hill-climbing, using all sorts of NetLogo modeling features: buttons, sliders, monitors, plots, and the graphics window. You've even written a quick and dirty procedure to give the turtles something to do. And that's where this tutorial leaves off.

You can continue with this model if you'd like, experimenting with different variables and algorithms to see what works the best (what makes the most turtles reach the peaks).

Alternatively, you can look at other models, or go ahead and build you own. You don't even have to model anything. It can be pleasant just to watch patches and turtles forming patterns, or whatever. Hopefully you will have learned a few things, both in terms of syntax and general methodology for model- building. The entire code as created above is shown below.

Appendix: Complete Code

The complete model is also available in NetLogo's Models Library, in the Code Examples section. It's called "Tutorial Example".

```
patches-own [ elevation ]      ;; elevation of the patch

globals [
  highest         ;; maximum patch elevation
```

```
  lowest          ;; minimum patch elevation
  turtles-moved?  ;; so we know when to stop the model
]

;; We also have two slider variables, 'number' and
;; 'smoothness'. 'number' determines the number of
;; turtles, and 'smoothness' determines how erratic
;; terrain becomes during diffusion of 'elevation'.

;; resets everything
to setup
  ca
  setup-patches
  setup-turtles
end

;; creates a random landscape of patch elevations
to setup-patches
  ask patches [set elevation (random 10000) ]
  repeat smoothness [diffuse elevation 1 ]
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]

  set highest max values-from patches [elevation]
  set lowest min values-from patches [elevation]
  ask patches [
    if (elevation > (highest - 100))
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black]
  ]
end

;; initializes the turtles
to setup-turtles
  crt number
  ask turtles [
    if (color = green) [ set color red ]
    fd (random screen-edge-x)
  ]
end

;; RUN-TIME PROCEDURES
;; main program control
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
  [ stop ]
end

;; performs a greedy-search of radius 1
;; for highest elevation
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
    [
      fd 1
      set turtles-moved? true
    ]
```

```
  ]
end

to do-plots
  set-current-plot "Turtles at Peaks"
  plot count turtles with
    [ elevation >= (highest - 100) ]
end
```

Interface Guide

This section of the manual walks you through every element of the NetLogo interface in order and explains its function.

In NetLogo, you have the choice of or viewing models found in the Models Library, adding to existing models, or creating your own models. The NetLogo interface was designed to meet all these needs.

The interface can be divided into two main parts: NetLogo menus, and the main NetLogo window. The main window is divided into tabs.

Menus

On Macs, if you are running the NetLogo application, the menubar is located at the top of the screen. On other platforms, and in the applet, the menubar is found at the top of the NetLogo window.

The functions available from the menus in the menubar are listed in the following chart.

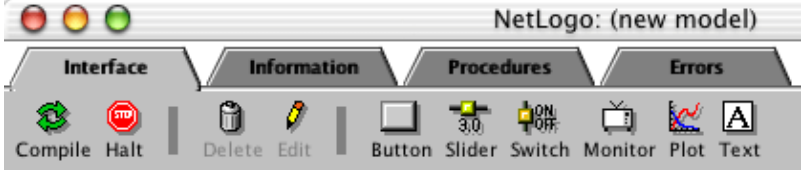
Chart: NetLogo Menus

	New	Starts a new model.
	Open	Opens any NetLogo or StarLogoT model on your computer.
	Models Library	A collection of demonstration models.
	Save & Save As	Used to save models.
	Save As Applet	Used to save a web page in HTML format that has your model embedded in it as a Java "applet".
	Print	Sends the contents of the currently showing tab to your printer.
	Export World	Saves all variables and the current state of all turtles and patches to a file.
	Export Plot	Saves the data in a plot to a file.
	Export Output	Save the contents of the output section of the command center to a file.
	Import World	Load a file that was saved by Export World.

	Quit	Exits NetLogo.
Edit		
	Undo	Undo that last action user performed.
	Cut	Cuts out or removes the selected text and temporarily saves it to the clipboard.
	Copy	Copies the selected text.
	Paste	Places the clipboard text where cursor is currently located.
	Delete	Deletes selected text.
	Find / Find Again	Finds a word or sequence of characters within the text windows. Find Again will find the next occurrence of the word or sequence.
Tools		
	Globals Monitor	Displays the values of all global variables.
	Patch Monitor	Displays the values of all of the variables in a particular patch. You can can also edit the values of the patch's variables and issue commands to the patch. (You can also open a patch monitor via the Graphics Window; see the Graphics Window section below.)
	Turtle Monitor	Displays the values of all of the variables in a particular turtle. You can can also edit the values of the turtle's variables and issue commands to the turtle. (You can also open a turtle monitor via the Graphics Window; see the Graphics Window section below.)
	Shapes Editor	Draw turtle shapes. See the Shapes Editor Guide for more information.
	BehaviorSpace	Runs the model over and over with different settings. See the BehaviorSpace Guide for more information.
Zoom		
	Larger	Increase the overall screen size of the model. Useful on large monitors or when using a projector in front of a group.
	Normal Size	Reset the screen size of the model to the normal size.
	Smaller	Decrease the overall screen size of the model.
Tabs		This menu offers keyboard shortcuts for each of the tabs. (On Macs, it's Command 1 through Command 4. On Window, it's Control 1 through Control 4.)
Help		
	About NetLogo	Information on the current NetLogo version the user is running. (On Macs, this menu item is on the Apple menu instead.)
	User Manual	Opens this manual in a web browser.

Main Window

At the top of NetLogo's main window are four tabs labeled "Interface", "Procedure", "Information", and "Errors". Only one tab at a time can be visible, but you can switch between them by clicking on the tabs at the top of the window.



Right below the row of tabs is a toolbar containing a row of buttons. The buttons available vary from tab to tab.

Interface Tab

The Interface Tab is where you watch your model run. It also has tools you can use to inspect and alter what's going on inside the model.

When you first open NetLogo, the Interface tab is empty except for the Graphics Window, where the turtles and patches appear, and the Command Center, which allows you to issue NetLogo commands.

Interface Toolbar

The toolbar contains all of the interface elements that are needed to display information in the Interface tab.

It is divided into sections with the left side containing two general controls that can be accessed from any tab: compile and halt. Compile is a command that filters through the procedure looking for any syntax errors. Halt is an emergency stopping mechanism used only when the model is stuck.

The other buttons in the toolbar are described below.

Working With Interface Elements

Selecting: To select an interface element, drag a rectangle around it with your mouse. A gray border will appear around the element to indicate that it is selected.

Selecting Multiple Items: You can select multiple interface elements at the same time by including them in the rectangle you drag. If multiple elements are selected, one of them is the "key" item, which means that if you use the "Edit" or "Delete" buttons on the Interface Toolbar, only the key item is affected. The key item is indicated by a darker gray border than the other items.

Unselecting: To unselect all interface elements, click the mouse on the white background of the Interface tab. To unselect an individual element, control−click (Macintosh) or right−click (other OS) the element and choose "Unselect" from the popup menu.

Editing: To change the characteristics of an interface element, select the element, then press the "Edit" button on the Interface Toolbar. You may also double click the element once it is selected. A third way to edit an element is to control−click (Macintosh) or right−click (other OS) it and choose "Edit" from the popup menu. If you use this last method, it is not necessary to select the element first.

Moving: Select the interface element, then drag it with your mouse to its new location. If you hold down the shift key while dragging, the element will move only straight up and down or straight left and right.

Resizing: Select the interface element, then drag the black "handles" in the selection border.

Deleting: Select the element or elements you want to delete, then press the "Delete" button on the Interface Toolbar. You may also delete an element by control−clicking (Macintosh) or right−clicking (other OS) it and choosing "Delete" from the popup menu. If you use this latter method, it is not necessary to select the element first.

To learn more about the different kinds of interface elements, refer to the chart below.

Chart: Interface Toolbar

Icon	Name	Description
	Button	Buttons can be either *once−only* buttons or *forever* buttons. When you click on a once−button, it executes its instructions once. The forever−button executes the instructions over and over, until you click on the button again to stop the action.
	Slider	Sliders are global variables, which are accessible by all agents. They are used in models as a quick way to change a variable without having to recode the procedure every time. Instead, the user moves the slider to a value and observe what happens in the model.
	Switch	Switches are a visual representation for a true/false variable. The user is asked to set the variable to either on (true) or off (false) by adjusting the switch.
	Monitor	Monitors display the value of any expression. The expression could be a variable, a complex expression, or a call to a reporter. Monitors automatically update several times per second.
	Plot	Plots are real−time graphs of data the model is generating.
	Text Box	The Text Box lets you create text labels in the Interface Tab.

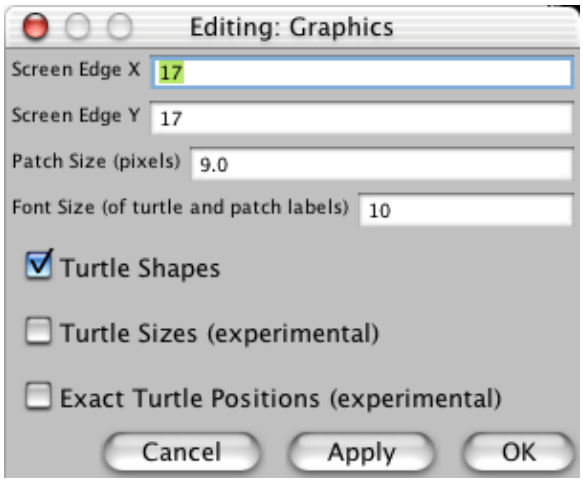
Graphics Window

The Graphics Window initially appears as a large black square on the Interface Tab. This is the graphical world of NetLogo's turtles and patches.

Some NetLogo models let you interact with the turtles and patches with your mouse by clicking and dragging in the Graphics Window.

The Graphics Window provides an easy way to open a turtle monitor or patch monitor. Just control−click (Macintosh) or right−click (other OS) on the turtle or patch you want to inspect, and choose "inspect turtle ..." or "inspect patch ..." from the popup menu. (Turtle and patch monitors can also be opened from the Tools menu or by using the `inspect` command.)

There are a number of settings associated with the Graphics Window. You can change these settings by editing the Graphics Window, as described in the "Working With Interface Elements" section above. Here are the settings for the Graphics Window:



To change the size of the Graphics Window adjust the "Patch Size" setting, which is measured in pixels. This does not change the number of patches, only how large the patches appear on the screen. To change the number of patches, alter the "Screen Edge X" and "Screen Edge Y" settings. (Note that changing the numbers of patches requires rebuilding the NetLogo world; you will lose all turtles and the values of all variables.)

In most NetLogo models, every turtle is always the same size: the same size as all the patches. The "Turtle Sizes" checkbox lets you turn on an optional feature where turtles can be drawn in all different sizes. The size of each turtle is controlled by its `size` turtle variable.

In most NetLogo models, every turtle is always drawn as if it were standing on the center of its patch. The "Exact Turtle Positions" checkbox lets you turn on an optional feature where turtles can be drawn in their exact positions within a patch.

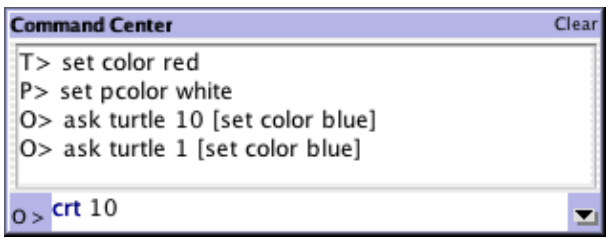
Note: the "Turtle Sizes" and "Exact Turtle Positions" features may make your model run a lot slower and/or cause display anomalies. We plan to improve these features in a future version of NetLogo.

Command Center

The Command Center allows you to issue commands directly, without adding them to the model's procedures. (Commands are instructions you give to turtles, patches, and the observer.) This is useful for inspecting and manipulating agents on the fly.

(Tutorial #2: Commands is an introduction to using commands in the Command Center.)

Let's take a closer look at the design of the Command Center.



You will notice there is a large display box, an agent popup menu (O>), a "clear" button, and the history popup menu (with the little black triangle). The top large display box temporarily stores all of

the commands that are entered into the Command Center. This area is strictly for reference; commands cannot be accessed or changed from this box. To clear this box, click "clear" in the top right corner.

The smaller text box, below the large box, is where commands are entered. On the left of this box is the agent popup menu, and on the right is the history popup menu.

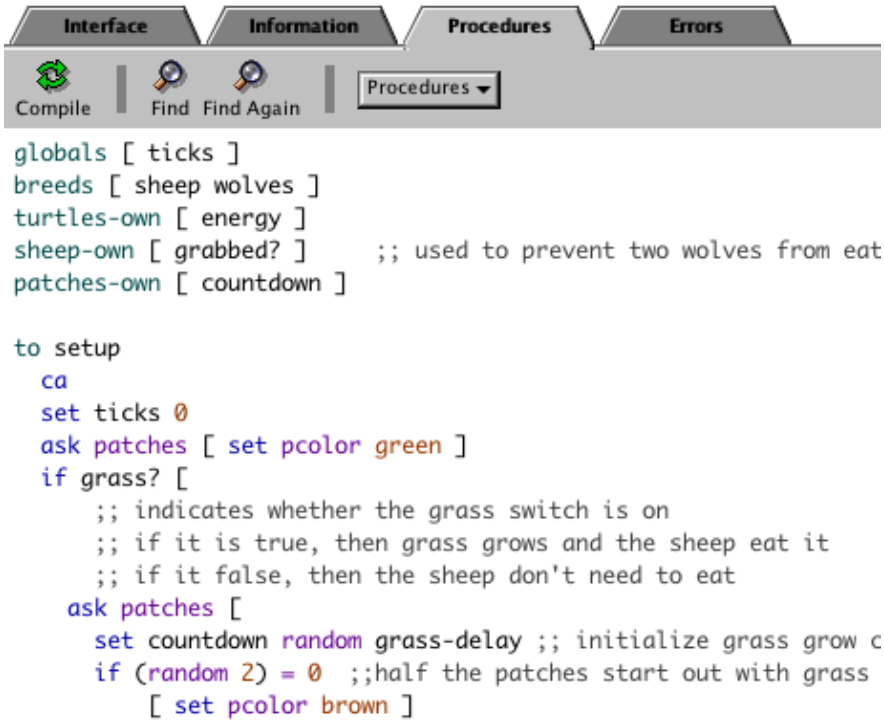
The agent popup menu allows you to select either observer, turtles, or patches. This is an easy way to assign an agent to a command and it is the same as writing `ask turtles [ ... ]`. Note: a quicker way to change between observer, turtles, and patches is to use the tab key on your keyboard.

The history popup menu lists all of the commands entered that can be accessed and used again. The up and down arrow keys on your keyboard will retrieve that last command that was written.

Note that pressing the "clear" function clears only the large display box and not the history. To clear the history section, choose "clear history", found at the top of its popup menu.

Procedures Tab

This tab is the workspace where the code for the model is stored. Commands you only want to use immediately go in the Command Center; commands you want to save and use later, over and over again, are found in the Procedures tab.



To determine if the code has any errors, press the "Compile" button. If there are any syntax errors, the Errors Tab will come to the front of the screen and turn red. The code that contains the error will be highlighted and a comment will appear in the top box. Switching tabs also causes the code to be compiled and any errors will be shown, so if you switch tabs, pressing the Compile button first isn't necessary.

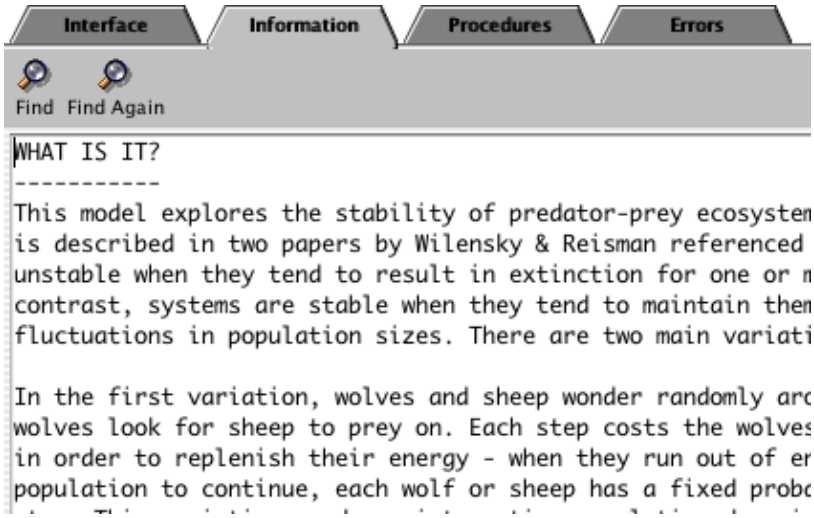
To find a fragment of code in the procedures, click on the magnifying glass on the Procedures Toolbar. Then enter the text you are looking for, and hit the "Find" button. The "Find Again" button finds the next location of the word or fragment of code throughout the procedure.

To find a particular procedure definition in your code, use the "Procedures" popup menu in the Procedures Toolbar. The menu lists all procedures in alphabetical order.

For more information about writing procedures, read Tutorial #3: Procedures and the Programming Guide.

Information Tab

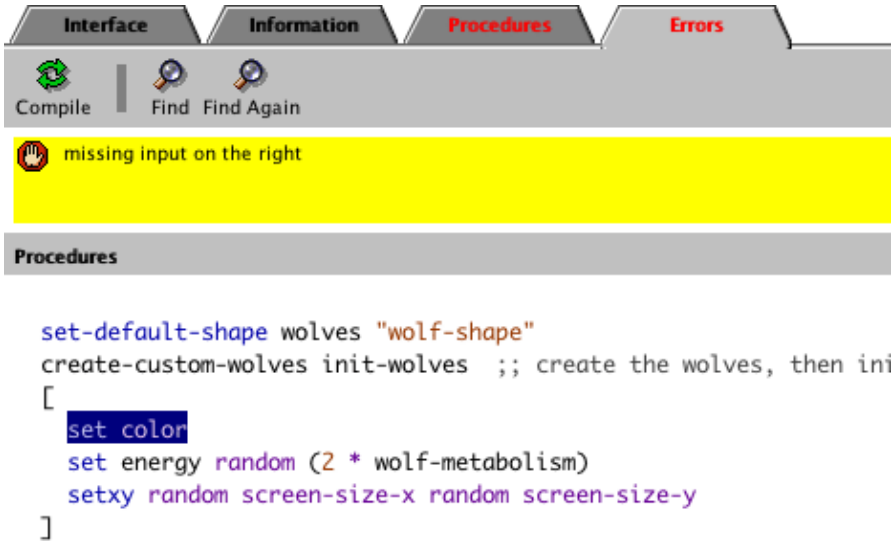
The Information tab provides an introduction to the model and an explanation of how to use it, things to explore, possible extensions, and NetLogo features. It is very helpful when you're first exploring a model.



We recommend reading the Information Tab before starting the model. The Information Tab explains what principle is being modeled and how the model was created.

Errors Tab

The Errors Tab provides a place for errors to be highlighted and brought to your attention. If there are any errors in your code, the "Errors" Tab will turn red and come to the front with an explanation of what caused the error.



Switching tabs, or pressing the "Compile" button in the toolbar, causes your code to be checked for errors.

Programming Guide

The following material explains some important features of programming in NetLogo.

(Note: If you are already familiar with StarLogo or StarLogoT, then the material in the first four sections may already be familiar to you.)

The Code Example models mentioned throughout can be found in the Code Examples section of the Models Library.

Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. Each agent can carry out its own activity, all simultaneously.

In NetLogo, there are three types of agents: turtles, patches, and the observer. Turtles are agents that move around in the world. The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move. The observer doesn't have a location −− you can imagine it as looking out over the world of turtles and patches.

When NetLogo starts up, there are no turtles yet. The observer can make new turtles. Patches can make new turtles too. (Patches can't move, but otherwise they're just as "alive" as turtles and the observer are.)

Patches have coordinates. The patch in the center of the world has coordinates (0, 0). We call the patch's coordinates `pxcor` and `pycor`. Just like in the standard mathematical coordinate plane, `pxcor` increases as you move to the right and `pycor` increases as you move up.

The total number of patches is determined by the settings `screen-edge-x` and `screen-edge-y`. When NetLogo starts up, both `screen-edge-x` and `screen-edge-y` are 17. This means that `pxcor` and `pycor` both range from −17 to 17, so there are 35 times 35, or 1225 patches total. (You can change the number of patches by editing NetLogo's Graphics window.)

Turtles have coordinates too: `xcor` and `ycor`. A patch's coordinates are always integers, but a turtle's coordinates can have decimals. For speed, NetLogo always draws a turtle on−screen as if it were standing in the center of its patch, but in fact, the turtle can be positioned at any point within

the patch.

The world of patches isn't bounded, but "wraps" −− so when a turtle moves past the edge of the world, it disappears and reappears on the opposite edge. Every patch has the same number of "neighbor" patches −− if you're a patch on the edge of the world, some of your "neighbors" are on the opposite edge.

Procedures

In NetLogo, commands and reporters tell agents what to do. **Commands** are actions for the agents to carry out. **Reporters** carry out some operation and report a result either to a command or another reporter.

Commands and reporters built into NetLogo are called **primitives**. The Primitives Dictionary has a complete list of built−in commands and reporters.

Commands and reporters you define yourself are called **procedures**. Each procedure has a name, preceded by the keyword `to`. The keyword `end` marks the end of the commands in the procedure. Once you define a procedure, you can use it elsewhere in your program.

Many commands and reporters take **inputs** −− values that the command or reporter uses in carrying out its actions.

Examples: Here are two command procedures:

```
to setup
  ca              ;; clear the screen
  crt 10          ;; make 10 new turtles
end

to go
  ask turtles
    [ fd 1              ;; all turtles move forward one step
      rt random 10      ;; ...and turn a random amount
      lt random 10 ]
end
```

Note the use of semicolons to add "comments" to the program. Comments make your program easier to read and understand.

In this program,

- `setup` and `go` are user−defined commands.
- `ca` ("clear all"), `crt` ("create turtles"), `ask`, `lt` ("left turn"), and `rt` ("right turn") are all primitive commands.
- `random` and `turtles` are primitive reporters. `random` takes a single number as an input and reports a random number that is less than the input (in this case, between 0 and 9). `turtles` reports the agentset consisting of all the turtles. (We'll explain about agentsets later.)

`setup` and `go` can be called by other procedures or by buttons. Many NetLogo models have a once−button that calls a procedure called `setup`, and a forever−button that calls a procedure called

go.

In NetLogo, you must specify which agents −− turtles, patches, or the observer −− are to run each command. (If you don't specify, the code is run by the observer.) In the code above, the observer uses `ask` to make the set of all turtles run the commands between the square brackets.

`ca` and `crt` can only be run by the observer. `fd`, on the other hand, can only be run by turtles. Some other commands and reporters, such as `set`, can be run by different agent types.

Variables

Variables are places to store values (such as numbers). A variable can be a global variable, a turtle variable, or a patch variable.

If a variable is a global variable, there is only one value for the variable, and every agent can access it. But each turtle has its own value for every turtle variable, and each patch has its own value for every patch variable.

Some variables are built into NetLogo. For example, all turtles have a `color` variable, and all patches have a `pcolor` variable. (The patch variable begins with "p" so you won't get it confused with the turtle variable.) If you set the variable, the turtle or patch changes color. (See next section for details.)

Other predefined turtle variables including `xcor`, `ycor`, and `heading`. Other predefined patch variables include `pxcor` and `pycor`. (There is a complete list here.)

You can also define your own variables. You can make a global variable by adding a switch or a slider to your model, or by using the `globals` keyword at the beginning of your code, like this:

```
globals [ clock ]
```

You can also define new turtle and patch variables using the `turtles-own` and `patches-own` keywords, like this:

```
turtles-own [ energy speed ]
patches-own [ friction ]
```

These variables can then be used freely in your model. Use the `set` command to set them. (If you don't set them, they'll start out storing a value of zero.)

Global variables can by read and set at any time by any agent. As well, a turtle can read and set patch variables of the patch it is standing on. For example, this code:

```
ask turtles [ set pcolor red ]
```

causes every turtle to make the patch it is standing on red.

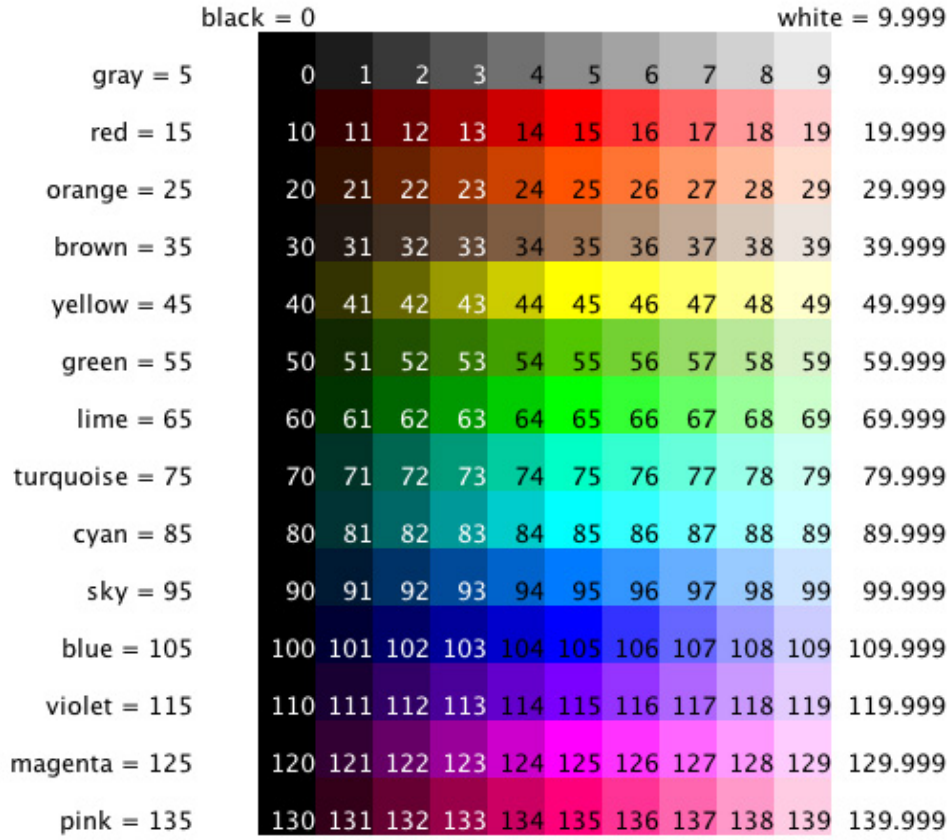
In other situations where you want an agent to read or set a different agent's variable, you put −of after the variable name and then specify which agent you mean. Examples:

```
set color-of turtle 5 red
;; turtle with ID number 5 turns red
```

```
set pcolor-of patch 2 3 green
;; patch with pxcor of 2 and pycor of 3 turns green
ask turtles [ set pcolor-of patch-at 1 0 blue ]
;; every turtle turns the patch to its east blue
ask patches with [any turtles-here]
  [ set color-of random-one-of turtles-here yellow ]
;; on every patch, a random turtle turns yellow
```

Colors

NetLogo represents colors as numbers in the range 0 to 140, with the exception of 140 itself. Below is a chart showing the range of colors you can use in NetLogo.



The chart shows that:

- Some of the colors have names. (You can use these names in your code.)
- Every named color except black and white has a number ending in 5.
- On either side of each named color are darker and lighter shades of the color.
- 0, 10, 20, and so on are all black. 9.999, 19.999, 29.999 and so on are all white.

Code Example: The color chart was made in NetLogo with the Color Chart Example model.

Note: If you use a number outside the 0 to 140 range, NetLogo will repeatedly add or subtract 140 from the number until it is in the 0 to 140 range. For example, 25 is orange, so 165, 305, 445, and

so on are orange too, and so are −115, −255, −395, etc. This calculation is done automatically whenever you set the turtle variable `color` or the patch variable `pcolor`. Should you need to perform this calculation in some other context, use the `wrap-color` primitive.

If you want a color that's not on the chart, more can be found between the integers. For example, 26.5 is a shade of orange halfway between 26 and 27. This doesn't mean you can make any color in NetLogo; the NetLogo color space is only a subset of all possible colors. A fixed set of discrete hues is available, and you can either decrease the brightness (darken) or decrease the saturation (lighten) of those hues to get your desired color, but you may not decrease both the brightness and saturation.

There are a few primitives that are helpful for working with color shades. The `scale-color` primitive is useful for converting numeric data into colors. And `shade-of?` will tell you if two colors are "shades" of the same basic hue. For example, `shade-of? orange 27` is true, because 27 is a lighter shade of orange.

> **Code Example:** Scale−color Example shows you how to use the scale−color reporter.

For many models, the NetLogo color system is a convenient way of expressing colors. But sometimes you'd like to be able to specify colors the conventional way, by specifying HSB (hue/saturation/brightness) or RGB (red/green/blue) values. The `hsb` and `rgb` primitives let you do this. `extract-hsb` and `extract-hsb` let you convert colors in the other direction.

Since the NetLogo color space doesn't include all hues, `hsb` and `rgb` can't always give you the exact color you ask for, but they try to come as close as possible.

> **Code Example:** You can use the HSB and RGB Example model to experiment with the HSB and RGB color systems.

Ask

NetLogo uses the `ask` command to specify commands that are to be run by turtles or patches. All code to be run by turtles **must** be located in a turtle "context". You can establish a turtle context in any of three ways:

- In a button, by choosing "Turtles" from the popup menu. Any code you put in the button will be run by all turtles.
- In the Command Center, by choosing "Turtles" from the popup menu. Any commands you enter will be run by all the turtles.
- By using `ask turtles`.

The same goes for patches and the observer, except that code to be run by the observer must not be inside any `ask`.

Here's an example of the use of `ask` syntax in a NetLogo procedure:

```
to setup
  ca
```

```
  crt 100              ;; create 100 turtles
  ask turtles
    [ set color red    ;; turn them red
      rt random 360    ;; give them random headings
      fd 50 ]          ;; spread them around
  ask patches
    [ if (pxcor > 0)         ;; patches on the right side
        [ set pcolor green ] ]  ;; of the screen turn green
end
```

The models in the Models Library are full of other examples. A good place to start looking is in the Code Examples section.

Usually, the observer uses `ask` to ask all turtles or all patches to run commands. You can also use `ask` to have make an individual turtle or patch run commands. The reporters `turtle`, `patch`, and `patch-at` are useful for this technique. For example:

```
to setup
  ca
  crt 3                  ;; make 3 turtles
  ask turtle 0           ;; tell the first one...
    [ fd 1 ]             ;; ...to go forward
  ask turtle 1           ;; tell the second one...
    [ set color green ]  ;; ...to become green
  ask turtle 2           ;; tell the third one...
    [ rt 90 ]            ;; ...to turn right
  ask patch 2 -2         ;; ask the patch at coords (2,-2)
    [ set pcolor blue ]  ;; ...to become blue
  ask turtle 0           ;; ask the first turtle
    [ ask patch-at 1 0   ;; ...to ask patch to the east
      [ set pcolor red ] ;; ...to become red
end
```

Every turtle created has an ID number. The first turtle created has ID 0, the second turtle ID 1, and so forth. The `turtle` primitive reporter takes an ID number as an input, and reports the turtle with that ID number. The `patch` primitive reporter takes values for pxcor and pycor and reports the patch with those coordinates. And the `patch-at` primitive reporter takes *offsets*: distances, in the x and y directions, *from* the first agent. In the example above, the turtle with ID number 0 is asked to get the patch east (and no patches north) of itself.

You can also select a subset of turtles, or a subset of patches, and ask them to do something. This involves a concept called "agentsets". The next section explains this concept in detail.

Agentsets

An agentset is exactly what its name implies, a set of agents. An agentset can contain either turtles or patches, but not both at once.

You've seen the `turtles` primitive, which reports the agentset of all turtles, and the `patches` primitive, which reports the agentset of all patches.

But what's powerful about the agentset concept is that you can construct agentsets that contain only *some* turtles or *some* patches. For example, all the red turtles, or the patches with pxcor evenly divisible by five, or the turtles in the first quadrant that are on a green patch. These agentsets can then be used by `ask` or by various reporters that take agentsets as inputs.

One way is to use turtles-here or turtles-at to make an agentset containing only the turtles on my patch, or only the turtles on some other particular patch.

Here are some more examples of how to make agentsets:

```
;; all red turtles:
  turtles with [color = red]
;; all red turtles on my patch
  turtles-here with [color = red]
;; patches on right side of screen
  patches with [pxcor > 0]
;; all turtles less than 3 patches away
  turtles in-radius 3
;; the four patches to the east, north, west, and south
  patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
;; shorthand for those four patches
  neighbors4
;; turtles in the first quadrant that are on a green patch
  turtles with [(xcor > 0) and (ycor > 0)
                and (pcolor = green)]
```

Once you have created an agentset, here are some simple things you can do:

- Use ask to make the agents in the agentset do something
- Use any to see if the agentset is empty
- Use count to find out exactly how many agents are in the set

And here are some more complex things you can do:

- Pick a random agent from the set using random-one-of. For example, we can make a randomly chosen turtle turn green:

  ```
  set color-of random-one-of turtles green
  ```

  Or tell a randomly chosen patch to sprout a new turtle:

  ```
  ask random-one-of patches [ sprout 1 [ ] ]
  ```

- Use the max-one-of or min-one-of reporters to find out which agent is the most or least along some scale. For example, to remove the richest turtle, you could say

  ```
  ask max-one-of turtles [sum assets] [ die ]
  ```

- Make a histogram of the agentset using the histogram command.
- Use values-from to make a list of values, one for each agent in the agentset. Then use one of NetLogo's list primitives to do something with the list. (See the "Lists" section below.) For example, to find out how rich the richest turtle is, you could say

  ```
  show max values-from turtles [sum assets]
  ```

This only scratches the surface -- see the Models Library for many more examples, and consult the Primitives Guide and Primitives Dictionary for more information about all of the agentset primitives.

More examples of using agentsets are provided in the individual entries for these primitives in the NetLogo Dictionary. In developing familiarity with programming in NetLogo, it is important to begin

to think of compound commands in terms of how each element passes information to the next one. Agentsets are an important part of this conceptual scheme and provide the NetLogo developer with a lot of power and flexibility, as well as being more similar to natural language.

Code Example: Ask Agentset Example

Breeds

NetLogo allows you to define different "breeds" of turtles. Once you have defined breeds, you can go on and make the different breeds behave differently. For example, you could have breeds called `sheep` and `wolves`, and have the wolves try to eat the sheep.

You define breeds using the `breeds` keyword, at the top of your model, before any procedures:

```
breeds [wolves sheep]
```

When you define a breed such as `sheep`, an agentset for that breed is automatically created, so that all of the agentset capabilities described above are immediately available with the `sheep` agentset.

The following new primitives are also automatically available once you define a breed: `create-sheep`, `create-custom-sheep` (`cct-sheep` for short), `sheep-here`, and `sheep-at`.

Also, you can use `sheep-own` to define new turtle variables that only turtles of the given breed have.

A turtle's breed agentset is stored in the `breed` turtle variable. So you can test a turtle's breed, like this:

```
if breed = wolves [ ... ]
```

Note also that turtles can change breeds. A wolf doesn't have to remain a wolf its whole life. Let's change a random wolf into a sheep:

```
ask random-one-of wolves [ set breed sheep ]
```

The `set-default-shape` primitive is useful for associating certain turtle shapes with certain breeds. See the section on shapes below.

Here is a quick example of using breeds:

```
breeds [mice frogs]
mice-own [cheese]
to setup
  ca
  create-custom-mice 50
    [ set color white
      set cheese random 10 ]
  create-custom-frogs 50
    [ set color green ]
end
```

Code Example: Breeds and Shapes Example

Synchronization

In both StarLogoT and NetLogo, turtle commands are executed asynchronously; each turtle does its list of commands as fast as it can. In StarLogoT, one could make the turtles "line up" by putting in a comma (,). At that point, the turtles would wait until all were finished before any went on.

The equivalent in NetLogo is to come to the end of an ask block. If you write it this way, the two steps are not synced:

```
ask turtles
  [ fd random 10
    do-calculation ]
```

But if you write it this way, they are:

```
ask turtles [ fd random 10 ]
ask turtles [ do-calculation ]
```

This latter form is equivalent to this use of the comma in StarLogoT:

```
fd random 10 ,
do-calculation
```

Procedures (advanced)

Here are some more advanced features you can take advantage of when defining your own procedures.

Procedures with inputs

Your own procedures can take inputs, just like primitives do. To create a procedure that accepts inputs, include a list of input names in square brackets after the procedure name. For example:

```
to draw-polygon [num-sides size]
   pd
   repeat num-sides
     [ fd size
       rt (360 / num-sides) ]
end
```

Elsewhere in the program, you could ask turtles to each draw an octagon with a side length equal to its ID-number:

```
ask turtles [ draw-polygon 8 who ]
```

Reporter procedures

Just like you can define your own commands, you can define your own reporters. You must do two special things. First, use `to-report` instead of `to` to begin your procedure. Then, in the body of the procedure, use `report` to report the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report 0 - number ]
end
```

Procedures with local variables

A local variables is defined and used only in the context of a particular procedure. To add a local variable to your procedure, use the `locals` keyword. It must come at the beginning of your procedure. For example:

```
to swap-colors [turtle1 turtle2]
   locals [temp]
   set temp color-of turtle1
   set (color-of turtle1) (color-of turtle2)
   set (color-of turtle2) temp
end
```

Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of information in a list. Each value in the list can be any type of value: a number, or a string, an agent or agentset, or even another list.

Lists allow for the convenient packaging of information in NetLogo. If your agents carry out a repetitive calculation on multiple variables, it might be easier to have a list variable, instead of multiple number variables.

Constant Lists

You can make a list by simply putting the values you want in the list between brackets, like this: `set mylist [2 4 6 8]`. Note that the individual values are separated by spaces. You can make lists that contains numbers and strings this way, as well as lists within lists, for example `[[2 4] [3 5]]`.

The empty list is written by putting nothing between the brackets, like this: `[]`.

Building Lists on the Fly

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the `list` reporter. The `list` reporter accepts two other reporters, runs them, and reports the results as a list.

If I wanted a list to contain two random values, I might use the following code:

```
set random-list list (random 10) (random 20)
```

This will set `random-list` to a new list of two random numbers each time it runs.

To make longer lists, use the `list` reporter with the `sentence` reporter, which concatenates two lists (combines their contents into a single, larger list).

The values−from primitive lets you construct a list from an agentset. It reports a list containing the each agent's value for the given reporter. (The reporter could be a simple variable name, or a more complex expression −− even a call to a procedure defined using `to−report`.) A common idiom is

```
max values−from turtles [...]
sum values−from turtles [...]
```

and so on.

Changing List Items

Technically, only one command changes a list −− `set`. This is used in conjunction with reporters. For example, to change the third item of a list to 10, you could use the following code:

```
set mylist [2 7 5 Bob [3 0 −2]]
; mylist is now [2 7 5 Bob [3 0 −2]]
set mylist replace−item 2 mylist 10
; mylist is now [2 7 10 Bob [3 0 −2]]
```

The `replace−item` reporter takes three inputs. The first input specifies which item in the list is to be changed. 0 means the first item, 1 means the second item, and so forth.

To add an item, say 42, to the end of a list, use the `lput` reporter. (`fput` adds an item to the beginning of a list.)

```
set mylist lput 42 mylist
; mylist is now [2 7 10 Bob [3 0 −2] 42]
```

But what if you changed your mind? The `but−last` (`bl` for short) reporter reports all the list items but the last.

```
set mylist but−last mylist
; mylist is now [2 7 10 Bob [3 0 −2]]
```

Suppose you want to get rid of item 0, the 2 at the beginning of the list.

```
set mylist but−first mylist
; mylist is now [7 10 Bob [3 0 −2]]
```

Suppose you wanted to change the third item that's nested inside item 3 from −2 to 9? The key is to realize that the name that can be used to call the nested list [3 0 −2] is `item 3 mylist`. Then the `replace−item` reporter can be nested to change the list−within−a−list. The parentheses are added for clarity.

```
set mylist (replace−item 3 mylist
                  (replace−item 2 (item 3 mylist) 9))
; mylist is now [7 10 Bob [3 0 9]]
```

The Primitives Dictionary has a section that lists of all the list−related primitives.

Strings

To input a constant string in NetLogo, surround it with double quotes.

The empty string is written by putting nothing between the quotes, like this: "".

Most of the list primitives work on strings as well:

```
butfirst "string" => "tring"
butlast "string" => "strin"
empty? "" => true
empty? "string" => false
first "string" => "s"
item 2 "string" => "r"
last "string" => "g"
length "string" => 6
member? "s" "string" => true
member? "rin" "string" => true
member? "ron" "string" => false
position "s" "string" => 0
position "rin" "string" => 2
position "ron" "string" => false
remove "r" "string" => "sting"
remove "s" "strings" => "tring"
replace-item 3 "string" "o" => "strong"
reverse "string" => "gnirts"
```

A few primitives are specific to strings, such as `is-string?`, `substring`, and `word`:

```
is-string? "string" => true
is-string? 37 => false
substring "string" 2 5 => "rin"
word "tur" "tle" => "turtle"
```

Strings can be compared using the =, !=, <, >, <=, and >= operators.

To concatenate strings, that is, combine them into a single string, you may also use the + (plus) operator, like this:

```
"tur" + "tle" => "turtle"
```

If you need to embed a special character in a string, use the following escape sequences:

- \n = newline (carriage return)
- \t = tab
- \" = double quote
- \\ = backslash

Turtle shapes

In StarLogoT, turtle shapes were bitmaps. They all had a single fixed size and could only rotate in 45 degree increments.

In NetLogo, turtle shapes are vector shapes. They are built up from basic geometric shapes; squares, circles, and lines, rather than a grid of pixels. Vector shapes are fully scalable and rotatable.

A turtle's shape is stored in its `shape` variable and can be set using the `set` command.

New turtles have a shape of "default". The `set-default-shape` primitive is useful for changing the default turtle shape to a different shape, or having a different default turtle shape for each breed of turtle.

Use the Shapes Editor to create your own turtle shapes. For more information, see the Shapes Editor section of this manual.

Code Examples: Breeds and Shapes Example, Shape Animation Example

HubNet Guide

HubNet is a technology that lets you use NetLogo to run *participatory simulations* in the classroom. In a participatory simulation, a whole class takes part in enacting the behavior of a system as each student controls a part of the system by using an individual TI−83+ calculator.

For example, in the Gridlock simulation, each student controls a traffic light in a simulated city with their calculator. The class as a whole tries to make traffic flow efficiently through the city. As the simulation runs, data is collected which can then be analyzed with the calculators afterwards.

For more information on participatory simulations and their learning potential, please visit the Participatory Simulations Project web site.

About HubNet

What do I need to get started?

- **A computer with an attached projector.** This computer will run NetLogo and project the simulation for class viewing.
- **A classroom set of TI−83+ graphing calculators.**
- **TI Navigator wireless calculator network.**

NOTE: Navigator is not yet commercially available. To learn more about the system, visit Texas Instruments' site. In the future, we hope to support other input devices such as laptops and PDA's (Personal Digital Assistants).

First−time NetLogo user?

NetLogo is a programmable modeling environment. It comes with a large library of existing simulations, both participatory and traditional, that you can use and modify. Content areas include social science and economics, biology and medicine, physics and chemistry, and mathematics and computer science. You and your students can also use it to build your own simulations, if you choose.

In traditional NetLogo simulations, the simulation runs according to rules that the simulation author specifies. HubNet adds a new dimension to NetLogo by letting simulations run not just according to

rules, but by direct human participation. Since HubNet builds upon NetLogo, we recommend that before trying HubNet for the first time, you should be familiar with the basics of NetLogo.

Teacher workshops

For information on up and coming workshops and NetLogo and HubNet use in the classroom, please contact us at feedback@ccl.northwestern.edu.

Getting Started With HubNet

Using NetLogo

We recommend that you become familiar with NetLogo itself before using the HubNet technology.

You can run NetLogo in either of two ways:

- Run the NetLogo application, which you can download and install from the NetLogo web site
- Run the NetLogo applet in your browser at our web site

The application starts up faster and has some extra features. However, if your school has firewalls or proxy servers, consult the "Proxy Web Servers & Firewalls" section below.

You can become familiar with NetLogo by trying out some of the models in the Models Library. Open the Models Library from the File menu in NetLogo. Then click on a model that you want to try and press the Open button. The Information tab in each of the models gives background information and instructions.

Other sections of the NetLogo User Manual may be helpful when learning NetLogo. We suggest that beginning users focus on the section Tutorial #1: Running Models.

If you have any questions about NetLogo, feel free to E−mail us. You can reach us at feedback@ccl.northwestern.edu.

HubNet Activities

Below are the current HubNet activities that are fully developed. For each model, you will find its educational goals and suggested ways to incorporate them into your classroom.

- Disease −− A disease spreads through the simulated population of students.
- Elevators −− Student−controlled elevators demonstrate the relation of velocity and position.
- Function Activity −− Students experience the concept of a function by forming themselves into graphs.
- Gridlock −− Students use traffic lights to control the flow of traffic through a city.
- People Molecules −− Using CBR's (Calculator Based Rangefinders), students use their bodies to represent gas molecules.
- Regression −− As students move on−screen, they see the best−fit line of their positions.

NOTE: In addition to the discussion of learning goals and classroom techniques, these materials also contain step by step instructions and screen shots. As of June 2002, they are out of date and no longer match the actual activities in many respects. We are working on updating them; we expect these materials to be updated by July 15, 2002. In the meantime, please use these materials for the discussions, but for step by step instructions, rely instead on the QuickStart Instructions built into the activities (see next section).

Running an activity

You'll find the HubNet activities in NetLogo's Models Library, under the HubNet Activities folder.

When you open the first HubNet activity for each session of NetLogo, you will be prompted by a login dialog. This prompts you to enter information (such as User Id and Password) that is necessary for connecting to the appropriate server and running the HubNet activity. If you don't actually want to run the model, just press the Cancel button. (Be aware you may still get NetLogo Runtime errors if you do press cancel.)

In each of the activities, you'll see a box on the screen labeled "QuickStart Instructions". It contains step by step instructions on how to run that activity. Click the "Next>>>" button to advance to the next instruction.

We suggest doing a few practice runs of the activity before trying it in front of an actual class.

If you have any questions about running the activities, feel free to E−mail us. You can reach us at feedback@ccl.northwestern.edu.

Proxy web servers and firewalls

If your school uses a proxy web server or a firewall, you must use the NetLogo applet, not the NetLogo application. (You may need to adjust the preferences in your browser to let it know about your proxy server or firewall.)

If you are not sure if your school uses a proxy web server or firewall, or you need help configuring your browser to have the correct proxy settings, ask your network administrator.

We are actively working to remove this restriction on the NetLogo application. We also expect that a future version of the TI Navigator system may help address the situation. If you have any trouble running HubNet in your classroom, please contact us at feedback@ccl.northwestern.edu.

HubNet Programming Guide

This section explains how to use NetLogo to modify the existing HubNet activities or build your own, new HubNet activities.

Calculator

The calculator is able to send and receive the following data types from NetLogo:

- Valid calculator lists, such as "L1" or "PLOTS"
- Valid calculator matrices, such as [A] or [B]

- Valid calculator strings, such as "Str1" or "Str5"
- Numbers, such as "A" or "B"

The calculator sends and receives data by storing a set of parameters in the string "Str0". Depending upon what type of data you are trying to send or receive, "Str0" will have different values. For instance, if the modeler wanted to create and send a list of numbers in the list "L1", it would be done as follows. Set the value of the list to some numbers (in this case, 20, A, and B where A and B are number variables that are set previously in the calculator code). Then write:

```
{20,A,B}->L1
"1 L1"->Str0
Asm(prgmSENDVAR)
```

The length of the list of numbers that a calculator sends depends on what information you want to send to the NetLogo model. Further, how those numbers are interpreted by the model is also up to you.

You can also receive data from the NetLogo model. To do this, use the following calculator code:

```
"4 Str6 1"->Str0
Asm(prgmGETVAR)
```

Let's take a look at how the values of the string "Str0" are set. The first input in the string represents the type of variable that you are trying to get. Since we are trying to get a string ("Str6"), we give the first input the value 4. (See below for the values of legal data types.) The second input is the variable in which you would like to save the data received. In this case, we want to save the data to the variable "Str6". The third input tells how you would like to save this data into this variable. (See below for values of valid commands.) It should be noted that for sending a variable, the command defaults to 0, i.e. no command.

data type	associated value
number	0
list of numbers	1
matrix of numbers	2
string	4

command number	command explanation
0	No Command
1	Collate (Lists into a matrix, reals into a list, append strings)
2	Teacher Variable
4	Append Lists

You should note that you must always save the information into the variable "Str0" when you are sending or receiving information from the calculators. You can't use any other variable.

For more information on writing the calculator program portion of a HubNet Activity, please read the on−line documentation provided by TI. It is under the Help link.

Saving

The data sent by calculators or NetLogo is saved in the order that the server receives the data.

NetLogo Commands

In the model the modeler uses a set of commands to extract the data from the server. The modeler is free to interpret these numbers as anything they wish. A 5 could mean to set a particular patch's color to red or have a turtle move forward.

Setup

In order for NetLogo to communicate with the server, it is necessary for NetLogo to establish a connection with the server and for NetLogo to tell the server what variables to send to NetLogo. This is done with the following three primitives:

`hubnet-reset`
: This logs you into the HubNet system. You must be logged in to use any of the other HubNet primitives. If this is the first time called for this NetLogo session, a HubNet properties dialog appears prompting you to input the appropriate information to be able to log into the HubNet system. Once you press the Login button, NetLogo will attempt to log you into HubNet. Once you are logged in, you are always logged in while this model is open; you log off when you close the model or quit NetLogo.

`hubnet-set-tags variable-list`
: This sets which variables NetLogo expects from the calculators. NetLogo will only check for these variables and will ignore all others. Currently, the valid types that NetLogo will be able to receive from the calculator are the following:
  - ◊ Valid calculator lists, such as "L1" or "PLOTS"
  - ◊ Valid calculator matrices, such as [A] or [B]
  - ◊ Valid calculator strings, such as "Str1" or "Str5"
  - ◊ Numbers, such as "A" or "B"

  This primitive must be called before you try to check for data on the server using the `hubnet-message-waiting?` reporter.

`hubnet-set-client-interface client-type activity-name`
: If *client-type* is "TI-83+", notify the user to enable the activity *activity-name* on the TI Navigator web site. Future versions of HubNet may support other client types, and/or change the meaning of the second input to this command.

These are usually best called from the `startup` procedure of the NetLogo model.

Data extraction

The data extraction primitives are:

`hubnet-message-waiting?`
: This looks for new information on the server. It returns TRUE if there is new data, and FALSE if there is not.

`hubnet-fetch-message`
: This retrieves any new data from the server, so that it can be accessed by `hubnet-message`. This will cause an error if no data is on the server. So be sure to check

for data with `hubnet-message-waiting?` before calling this.

`hubnet-message-source`
: This reports the user ID that sent the data. This will cause an error if no data has been fetched from the server. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

`hubnet-message-tag`
: This reports the variable name that was sent. This will only report one of the tags set with the `hubnet-set-tags` primitive. This will cause an error if no data has been fetched from the server. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

`hubnet-message`
: This reports the data collected by `hubnet-fetch-message`. This will cause an error if no data has been fetched from the server. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

Sending data

It is also possible to send data from NetLogo to the server to be accessed by the calculators.

Note: It is not currently possible to send data from NetLogo directly to only an individual calculator. However, once the server has the data, any connected calculator can grab it. This is done using the calculators' communication facilities, rather than through NetLogo.

The primitive for sending data to the server is:

`hubnet-broadcast variable-name value`
: This broadcasts *value* from NetLogo to the variable *variable-name* on the server. You may send a number, a string, a list of numbers, or a matrix (a list of lists) of numbers.

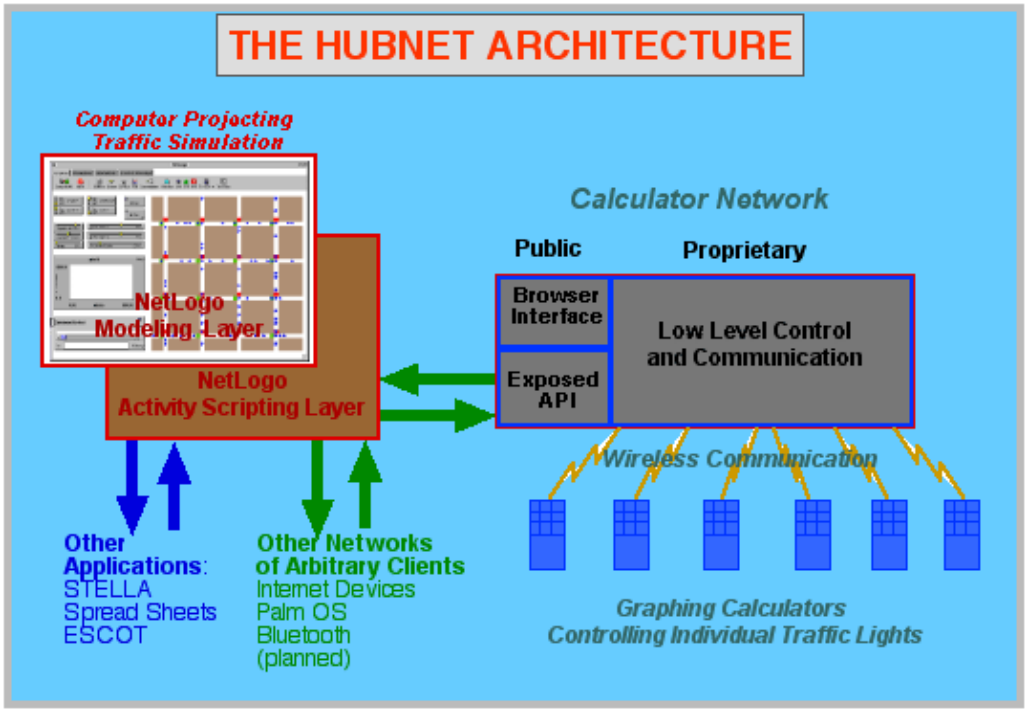
The examples of sending the various types of data that you can send are as follows:

data type	example
number	`hubnet-broadcast "A" 3.14`
string	`hubnet-broadcast "STR1" "HI THERE"`
list of numbers	`hubnet-broadcast "L2" [1 2 3]`
matrix of numbers	`hubnet-broadcast "[A]" [[1 2] [3 4]]`

Examples

Study the models in the "HubNet Activities" section of the Models Library to see how these primitives are used in practice in the Procedures window. Function Activity is a good one to start with.

Appendix: HubNet Architecture


THE HUBNET ARCHITECTURE
Computer Projecting
Traffic Simulation
NetLogo
Modeling Layer
NetLogo
Activity Scripting Layer
Calculator Network
Public
Proprietary
Browser
Interface
Exposed
API
Low Level Control
and Communication
Wireless Communication
Other
Applications:
STELLA
Spread Sheets
ESCOT
Other Networks
of Arbitrary Clients
Internet Devices
Palm OS
Bluetooth
(planned)
Graphing Calculators
Controlling Individual Traffic Lights

Shapes Editor Guide

The Shapes Editor allows you to create and save turtle designs. NetLogo uses fully scalable and rotatable vector graphics, which means it lets you create designs by combining basic geometric figures, which can appear on−screen in any size or orientation.

Getting Started

To begin making shapes, choose "Shapes Editor" on the Tools menu. A new window will open with a list of all the shapes, beginning with "default", and a row of action buttons, including new, edit, copy, and import.

Creating and Editing Shapes

Press the "new" button, and the editing window will appear. The shape you make will appear in the main drawing area and in the three smaller preview areas found on the left side of the editing window.

The preview areas show your shape at different sizes as it might appear within your model, as well as how it looks while rotating. The "rotation" feature can be turned off if you want a shape that always stays right side up.

There are four shape−drawing tools (select from four button icons), each of which can be filled with color or not, and a background grid to guide you. You can move and size these shapes with the mouse pointer as a drawing tool.

The shape created last will fall on top. Layers can be removed with the "remove last" button. The "remove last" and "remove all" buttons are the only way of correcting mistakes −− you can't go back to move or alter the shape after it's been drawn. If you make a mistake, to fix it, remove and redraw the shape.

Areas that use the "key" color (selected from a drop−down menu −− the default is gray) will change according to the value of each turtle's "color" variable in your model. Areas filled with any other color (selected from the palette at the top) will stay that color regardless of each turtle's "color". For example, you could create cars that always have yellow headlights and black wheels but have bodies of different colors −− to make this happen, you would draw the bodies in the "key" color.

It's tempting to draw complicated, interesting shapes, but remember that in most models, the patch size is so small that you won't be able to see all the detail. Simple, bold shapes are best.

When the shape is done, give it a name and click on the save option at the bottom of the screen. The shape and its name will now be in the list along with the "default" shape.

If you want to borrow a shape from another model, use the "import" button to call up other NetLogo models and import shapes from them.

Using Shapes in a Model

After closing the shapes editor, you can adjust the code to call the new shape. For example, suppose you want to create 50 turtles with the shape "rabbit". In the command center, give this command to the observer:

```
crt 50
```

And then give these commands to the turtles to spread them out, then change their shape:

```
fd random 15
set shape "rabbit"
```

Voila! Rabbits! Note the use of quotation marks around the shape name.

The `set-default-shape` primitive is also useful for assigning shapes to turtles.

BehaviorSpace Guide

This guide is broken up into three parts:

- **What is BehaviorSpace?**: A general description of the tool, including the ideas and principles behind it.
- **How it Works**: Walks you through how to use the tool and highlights its most commonly used features.
- **Interface Reference**: Offers comprehensive descriptions of each main window, as well as the interface components within those windows.

What is BehaviorSpace?

BehaviorSpace is a software tool integrated with NetLogo that allows you to perform experiments with models. It systematically varying the values of sliders and records the results of each corresponding model run. This way you can explore the model's "space" of possible behaviors and determine which combinations of slider values cause the behaviors of interest.

The need for this type of experiment is revealed by the following observations: Models often have more than one slider, each of which can take a large range of values. Together they form what in mathematics is called a parameter space for the model, whose dimensions are the number of sliders, and in which every point is a particular combination of values. Running a model with different slider values (and sometimes even the same ones) can lead to drastically different behavior in the system being modeled. So, how are you to know which particular configuration of values, or types of configurations, will yield the kind of behavior you are interested in? This amounts to the question of where in its huge, multi−dimension parameter space does your model perform best?

Suppose you want speedy synchronization from the agents in the Fireflies model. The model has four sliders −− number, cycle−length, flash−length and number−flashes −− that have approximately 2000, 100, 10 and 3 possible values, respectively. That means there are 2000 * 100 * 10 * 3 = 600,000 possible combinations of slider values! Trying combinations one at a time is hardly an efficient way to learn which one will evoke the speediest synchronization.

BehaviorSpace offers you a much better way to solve this problem. If you specify a subset of values from the ranges of each slider, it will run the model with each possible combination of those values and, during each model run, record the results. In doing so, it samples the model's parameter space −− not exhaustively, but enough so that you will be able to see relationships form between different sliders and the behavior of the system. Once all the runs are over, you are presented with a visual display with which you can explore the data from all the runs and determine how well each exhibited the behavior you are investigating.

The idea behind BehaviorSpace is that the way to truly understand a model is to run it multiple times with different parameter (slider) settings in order to see the whole range of behaviors the system is capable of producing. Only then is it possible to investigate when and why certain behaviors arise. Isolated trials are insufficient for this purpose because you have no reason to assume that the model will always demonstrate the particular behaviors you see. It's like eating in one restaurant in New York and then claiming you've seen all that the city has to offer. By enabling you to explore the entire parameter space of a model, and thus its entire space of behaviors,

BehaviorSpace can be a powerful tool for model understanding.

How It Works

You begin using BehaviorSpace by setting up an experiment. Its purpose is to run the model many times and collect data to determine which run best exhibits the behavior you are after. In the case of Fireflies, as mentioned above, data might be collected to reveal how quickly synchronization happened in each run.

Setting up an experiment initially entails specifying in the **slider values dialog** what possible values each slider in the model should take in the experiment, either in the form of a range or an arbitrary list, which is then displayed in the **sliders list**.

You can also dictate how long each run should last, either by entering the number of times the forever button should be called in the **time clicks text−box**, or by providing a reporter in the **exit condition text−area** that causes the run to stop when it reports a boolean value of true.

Next, you must specify a **behavior reporter**, which is the data you want continually recorded during a model run. For Fireflies, this would be the current extent of synchronization in the model, say the number of fireflies currently lit up, which you could enter with the following reporter:

```
count turtles with [color = yellow]
```

By collecting this data repeatedly throughout a run, an overall assessment can later be made as to how quickly synchronization was achieved.

Finally, you may select a **setup button** to initialize the model and the **forever button** that continues it.

By inputting the above information, you have told the system enough that it can run an experiment. As the experiment proceeds, it will produce data about each model run based on the behavior reporter you gave it.

In its raw form, this data is dry and difficult to interpret −− it is just long series of numbers. BehaviorSpace's job makes the data intuitively accessible by presenting it visually. It does so by allowing you to hold the values of all sliders but two constant, then presenting you with a grid of squares, called a **fitness landscape**.

The axes of the fitness landscape represent the two selected sliders. Each square in the landscape then represents the particular run whose slider values are determined by the location of the square along the two axis and the current values of the sliders held constant.

> **Example:** Suppose you are performing an experiment with the Fireflies model that tried many possible combinations of slider values. Then you select flash−length and number−flashes as the sliders to represent the axes of the landscape, with number held constant at 1500 and cycle−length at 100. Assuming flash−length is on the horizontal axis and number−flashes is on the vertical axis, the landscape would be 10 by 3, since flash−length goes from 1 to 10 and number−flashes from 1 to 3. Each square on the landscape would be colored a particular shade of blue, ranging from

solid blue to black, depending on how good (i.e. "fit") the behavior of the run represented by that square was according to criteria you select. For instance, squares representing runs of Fireflies in which the agents synchronized their lights quickly would be more blue than those representing runs where the fireflies synchronized slowly or not at all.

The reason the grid is called the "fitness landscape" is because it displays a 2−dimensional landscape of squares, each with a fitness, allowing you to take in with a glance regions of high and low fitness.

The criteria with which you rank run data −− and thereby determine fitness −− can be based on simple **point statistics**, such as maximum or mean value, or by **slope**, which is sensitive to the data's behavior (not to be confused with the model's behavior) over time.

In addition to changing the values of the constant sliders and alternating which sliders get to be on the axes of the landscape (these two operations let you explore the model's parameter space), you can see the data from each run represented on the landscape plotted on a **behavior plot** by moving your mouse over that run's corresponding square.

You can learn more about how evaluation of the experimental data works in the evaluation window part of the Interface Reference section below.

Interface Reference

This section describes the interface to BehaviorSpace. Included is a general description of each of the three main BehaviorSpace windows, a list of the interface components they contain, as well as a description of those components. It is intended mainly as a reference. The windows are discussed in the order you use them: first the Setup Window, then the Progress Dialog, and finally the Evaluation Window.

Setup Window

This is the window in which you set up an experiment. Setting up an experiment requires entering information in the fields outlined below, then pushing the **run comparison button**. When opening and closing models, or changing the existing model, the setup window is automatically updated, so there is no need to close and reopen it.

Component descriptions

Sliders list: The current list of sliders in the model, which is updated automatically as sliders are added, changed or deleted. The values specified to the right of each slider are the current values that slider is assigned to take in the experiment.

Change values button: Brings up the **Slider values dialog** so you can change slider values.

Slider values dialog: The dialog that pops up when the **change values button** is pushed.

- To specify values in a list, enter the values in the upper text−field. Values in the list can be separated by one or more spaces, commas, tabs, or semicolons.

- To specify values in a range, enter the first and last values the slider should take and the increment by which to traverse this range.
- First and last values of a range are inclusive.
- The first value must be less than or equal to the last one.
- The increment must be greater than zero.
- If a single value is entered more than once, then each instance will be treated as a separate value −− that is, if a particular slider has four 5's, then that slider will have the value 5 four times.
- To leave the dialog, there must be valid values in both the list and range text−fields.
- The values used for a particular slider in an experiment will be those whose checkbox is currently selected. For example, if the values range checkbox is selected, then the values specified by the range will be used, not those in the list.

Behavior reporter text area: A place to enter a NetLogo reporter to specify what kind of data is to be collected during each model run. This reporter is evaluated before every time click, or call to the forever button, and once at the end of the run, and its value is recorded for later analysis.

- Input must be a valid NetLogo reporter.
- Can use any reporters, whether built into the NetLogo language or written in the model.
- It must report a number.

Example: If you are interested in the population of sick people in the Virus model, you might want your behavior reporter to report how many sick people there currently are in the model with the following behavior reporter:

```
count people with [sick?]
```

Or, if you are investigating the distance between each turtle and the center of the screen in Turtles Circling, you might have it report the mean of all the turtle's distances to the center:

```
mean values−from turtles [distancexy 0 0]
```

Time clicks checkbox and text−field: When the checkbox is selected, each model run will end after the number of time clicks specified in the text−field. (One time click means one push, or iteration, of the forever button.)

- Input to the text−field must be a valid integer, *not* just any numeric reporter.

Exit condition checkbox and text−field: When the checkbox is selected, each model run will continue until the reporter entered in the text−field reports true. This reporter is run after every time click. The exit condition can be any valid NetLogo reporter that evaluates to true or false, such as those that use the operators =, <, >, <=, or >=

Example: The following exit condition causes model runs of Wolf−Sheep Predation to stop after either population is extinct:

```
(not any sheep) or (not any wolves)
```

Setup and forever button menus: Allow you to select the optional setup button and mandatory

forever button for an experiment. The setup button, if selected, is called once at the beginning of a model run, while the forever button is called repeatedly to run the model.

- To forgo the use of a setup button in an experiment, select the "<none>" option in the setup button menu.
- If you choose, the same button can be used as both the setup and forever buttons.
- If either the setup or forever button is chosen to be a forever button in your model, then one call to it (i.e. one time click) amounts to one call to its code, *not* to repeated calling of that code, just as if the actual button had been pressed.

Run comparison button: Begins the experiment, which is based on the input to the interface elements listed above.

Progress Dialog

This dialog appears automatically when an experiment begins to show you its progress. It contains real−time information about the current model run and the experiment as a whole. (It isn't necessary that you always watch model runs as they progress, but it may be helpful to monitor the progress of the experiment.) The dialog can be exited safely by hitting the **cancel button**. This will abort the experiment. If the experiment is allowed to run to completion, then this dialog disappears and is replaced by the Evaluation Window.

Component descriptions

Graphics window: Displays the state of the current model run.

- Has a fixed size, and does not allow for more than 300 patches across in either direction
- Is completely equivalent to the graphics window in the main interface in all other respects

Graphics on? checkbox: When unselected, graphics are temporarily turned off, although the model proceeds normally.

- Turning graphics off can make your experiment run significantly faster
- Can be selected and unselected repeatedly during an experiment

Current run number / slider values / time click text: Displays the number of the current run being performed, the values of the sliders in that run, and the number of time clicks that have elapsed

Behavior plot: Plots the value of the **behavior reporter** as the model run progresses.

Cancel button: Aborts the experiment, at which point the **progress dialog** quits.

Evaluation Window

The evaluation window is perhaps the most complex part of BehaviorSpace, but also the most flexible and powerful. It is designed to allow you to visually explore the behavior data reported by the model runs.

It works by holding the values of all but two sliders constant, then drawing a grid of squares, called a fitness landscape, that represents all possible value combinations of the last two. The values held

constant by the former sliders can be changed, and any of these sliders can be swapped in for one of the two sliders representing the axes on the grid. In this way you are essentially focusing on two sliders at a time, seeing what relationships exist between their values and the behavior of the model.

At the top of the window is a collection of radio boxes and pull−down menus, through which you can select criteria to rank the data reported by the different runs. These criteria can be changed as often as you wish, enabling you to highlight more than just one property of the data. Data can be ranked by slope characteristics, such as how much it increases over time, or the extent to which it can be fit to an exponential curve; or, the data can be ranked by a point statistic, such as its mean or minimum value.

Finally, you are able to mouse−over the landscape, and as the mouse passes over each square, which represents a single model run, the data from that run is plotted on the behavior plot to the left. Every time an experiment is run, a new evaluation window is created, so there can potentially be many on the screen at once. Each, however, only represents the data from the experiment that created it, so even if you change the name or number of sliders in your model, existing windows will remain unchanged.

Component descriptions

Point statistic checkbox: When this is selected, the data obtained from each run will be ranked according to the items selected by the following two menus. All three slope menus are made irrelevant. In general, it means that the data will be judged based on a simple, global statistic, instead of how it varies over time.

Point statistic type menu: Determines whether run data should be ranked according to their minimum, maximum, mean, or last value, or by the number of time clicks they lasted for. The latter option is only relevant when an exit condition was entered to stop runs, otherwise they will all finish in the same number of time clicks.

Point statistic value menu: Determines whether it is "better" for the statistic selected in the **point statistic type menu** to be high, low, or near 0.

- Low means more negative, so −100 is lower than 1
- Near 0 means as close to 0 as possible, irrespective of sign, so −.5 is nearer to 0 than 1

Slope checkbox: When selected, the data from each run will be ranked according to how it behaves over time, as specified by the following three menus. The items selected by the previous two menus are then irrelevant.

Slope type menu: Controls whether the run data is fit to a line or exponential curve. This distinction is only relevant with regards to error, since the slope of the data always remains the same. The higher the weight selected in the **relative weight menu**, the less this menu selection matters, because that means slope matters increasingly more than error. Error is determined by attempting to fit the data to either a line or a curve, depending on which is chosen, and then finding the Least−Square Error involved in doing so.

- Fitting data to a line means finding a function of the form y = ax + b that best approximates the data.

- Fitting data to an exponential curve means finding a function of the form y = ae^(bx), where e~=2.71828, that best approximates the data.
- If 'exponential' is selected and any run has both positive and negative data values, then a warning dialog is brought up saying that the data cannot be fit to an exponential curve, and 'linear' is automatically selected −− this occurs because exponential curves of this form can be either above the x−axis (when a is positive) or below the x−axis (when a is negative), but not both.

Slope value menu: Determines whether it is "better" for the data to be increasing (the more positive the slope the better), decreasing (the more negative the better), or constant (the closer to 0 the better)

Relative weight menu: Determines the relative importance of the slope of the data compared to how well it fits a line or exponential curve (see **slope type menu**). If 60 is chosen, then 60% of the data's rank comes from its slope (how much it is increasing, decreasing or constant) and 40% comes from line and curve−fitting error. So, a value of 100 means only slope is considered, while a value or 0 means only the error is considered.

- If you do not care about seeing strictly linear or exponential behavior in the run data, you should use high relative weight values, and otherwise use low ones.

Fitness landscape: Consists of a grid of squares, each of which represents the model run whose slider values were the current values of the constant sliders below and the ith and jth values of the two sliders on the grid's axes, where 'i' is the horizontal index of the square and 'j' is the vertical. The entire grid thus represents *all* runs in which the constant sliders had their current values (the current values being the ones displayed on the **set of slider components held constant** below the grid). This grid earns the name fitness landscape because the squares of which it is composed each has a fitness value, expressed by the color of the square, enabling you to see a landscape of colors denoting regions of high and low fitness −− that is, sets of neighboring near−black and near−blue squares. The landscape represents a 2−dimensional plane in the multi−dimensional parameter space of the model, since it accounts for all possible values of two sliders, while the rest are held constant. By changing the values of any constant sliders you are shifting that 2−D plane through the parameter space, and by selecting new sliders to be the axes, you are selecting a new plane entirely (that is, a plane through different dimensions).

- As you move your mouse over different squares in the grid, the data from those runs will appear in the **behavior plot** to the left, in addition to the line or curve that best fits the data if the **add best−fit−line checkbox** is selected.
- As your mouse passes over a square in the landscape, information about the model run that square represents appears above the landscape. This information consists of the value that each of the two selected sliders had during that run, and the overall fitness of that run.
- The fitness of a run is an estimation of how "good" the run was according to the criteria you select in the menus listed above. When the **slope checkbox** is selected, fitness derives from the slope of the data as well as how well it fits a linear or exponential curve. You can control the relative importance of these factors with the **relative weight menu**. When the **point statistic checkbox** is selected, the value that you see above the landscape is not the fitness of the run, but the value of the chosen statistic. For instance, the minimum value of the run is shown when "minimum value" is selected in the **point statistic type menu**. This value is *not* necessarily the fitness of the run, however, since the "goodness" of a particular value changes when you make a new selection in the **point statistic value menu**.

- The grid has a maximum height and width, which is reached when there are more than 5 squares in either direction, and cannot be resized.
- Squares turn gray when there is insufficient information to construct a grid −− this happens when either axis has not been assigned a slider (one axis is sufficient if a model only has one slider).

Set of sliders components held constant: These sliders consist of all those not present as axes of the **fitness landscape**. Since all the model runs represented on the landscape have the same value for these sliders −− namely, the one currently selected on the slider components −− they are considered held constant for the landscape. By changing the value of a single one of these sliders the whole landscape changes, because it now represents a whole new set of runs. If you changed a slider from 10 to 12, say, then all the runs in which that slider had the value 10 are replaced by those in which it had the value 12.

- Changing the value of these sliders does *not* affect the value of the sliders on the main interface.
- The only values the sliders can take are those that they actually had in the experiment.

Behavior plot: Displays the data from individual runs as you move the mouse over their corresponding squares on the **fitness landscape**. It's x−axis is time clicks and its y−axis is behavior −− that is, the value reported by the **behavior reporter** entered in the setup window.

- The best fit line or curve can be added along with the data if the **add best−fit−line checkbox** is selected.
- Run data can be continually collected in the plot and superimposed on each other by selecting the **superimpose plots checkbox**.
- Only 14 plot pen colors are used, so if more than 14 data sets are superimposed at a single time, the system will begin reusing colors for the new pens.
- Automatically scales to show all data.

Add best−fit−line checkbox: Adds the line or exponential curve that best fits the data to the **behavior plot**, depending on whether 'linear' or 'exponential' was selected in the **slope type menu**.

Superimpose plots checkbox: Stops old data from being removed from the **behavior plot** when new data is added, resulting in increasing sets of data being superimposed on each other. Unselected, the checkbox causes all data to be removed.

Export Plot button: Saves the data currently being displayed in the **behavior plot** to a file.

Export Behavior Data button: Saves the behavior data gathered during all of the model runs to a file. The data saved for each model run includes: the slider settings for that run; how many time ticks the run lasted; the minimum, maximum, average, and final value for the behavior reporter; and the value of the behavior reporter at each time tick during the run. (Note that the fitnesses are not exported, only the raw behavior data. In a future version of BehaviorSpace the fitness data will be exported as well.)

Note: The export buttons create files in plain−text, "comma−separated values" (`.csv`) format. CSV files can be read by most popular spreadsheet and database programs as well as any text editor.

FAQ (Frequently Asked Questions)

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

General

- What is NetLogo written in?
- How do I cite NetLogo in an academic publication?
- How do I cite a model from the Models Library in an academic publication?
- What license is NetLogo released under? Are there are any legal restrictions on use, redistribution, etc.?
- Is the source code to NetLogo available?
- Do you offer any workshops or other training on NetLogo?
- Has anyone written a model of <x>?
- What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?

Downloading

- The download form doesn't work for me. Can I have a direct link to the software?
- Downloading NetLogo takes too long. Can I get it some other way, like on a CD?
- I downloaded and installed NetLogo but the Models Library has few or no models in it. What's going on?

Usage

- How do I change how many patches there are?
- How big can my model be? How many turtles, patches, procedures, buttons, etc.?
- When will NetLogo be a native app on Mac OS X?
- Can I import a graphic into NetLogo?
- My model runs slowly. How can I speed it up?
- I want to try HubNet. Can I?
- Can I run a NetLogo model from the command line? Can I run it without a GUI?

Programming

- How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?
- The NetLogo world is a torus, that is, the edges of the screen are connected to each other, so turtles and patches "wrap around". Can I use a different world topology: bounded, infinite plane, sphere, etc.?
- Does NetLogo have a command like StarLogo's "grab" command?
- I tried to put `-at` after the name of a variable, for example `variable-at -1 0`, but NetLogo won't let me. Why not?
- I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?
- Is there some simple way to multiply one list by another list or number without writing a procedure?
- Can I read data from a file?

General

What is NetLogo written in?

NetLogo is written entirely in Java (version 1.1).

How do I cite NetLogo in an academic publication?

NetLogo itself: Wilensky, U. 1999. NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer−Based Modeling, Northwestern University. Evanston, IL.

HubNet: Wilensky, U. Stroup, W., 1999. HubNet. http://ccl.northwestern.edu/ps/. Center for Connected Learning and Computer−Based Modeling, Northwestern University. Evanston, IL.

How do I cite a model from the Models Library in an academic publication?

Wilensky, U. (year). Name of Model. URL of model. Center for Connected Learning and Computer−Based Modeling, Northwestern University. Evanston, IL.

To determine the URL for a model, visit our web−based version of the Models Library and click on the name of the model. An example model URL is: http://ccl.northwestern.edu/netlogo/models/PepperedMoths.

To determine the year, open the model from the NetLogo application and look in the copyright information at the bottom of the Procedures tab.

What license is NetLogo released under? Are there are any legal restrictions on use, redistribution, etc.?

The NetLogo software, models and documentation are distributed free of charge for use by the public to explore and construct models. Permission to copy or modify the NetLogo software, models and documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright notice and the original author's name appears on all copies and supporting documentation. For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission must be obtained from Uri Wilensky. The software, models and documentation shall not be used, rewritten, or adapted as the basis of a commercial software or hardware product without first obtaining appropriate licenses from Uri Wilensky. We make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

We are in the process of reevaluating the language of the license in response to user feedback. In the near future, we intend to send out a revised license.

Is the source code to NetLogo available?

At present, no. We are evaluating how best to distribute NetLogo when it is in a more mature state. Making the source available is one possibility.

We do understand, however, that it is important that NetLogo not be a closed and non−extensible platform. That is not our intention for the product. For a future version of NetLogo, we plan to add an API so that users can add their own extensions to NetLogo. We'd like to work with interested users to help design this API to suit the needs of our user community.

Do you offer any workshops or other training on NetLogo?

From time to time. Contact us at feedback@ccl.northwestern.edu if interested.

Has anyone written a model of <x>?

The best place to ask this question is on the NetLogo Users Group.

You should also check the NetLogo User Community Models section of our Models Library web page.

What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?

The first version of StarLogo was developed at the MIT Media Lab in 1989−90. This original version ran on a massively parallel supercomputer called The Connection Machine. A few years later (1994), a simulated parallel version was developed for the Macintosh computer. That version eventually became MacStarLogo. The StarLogoT (1997) software, developed at the Center for Connected Learning and Computer−Based Modeling (CCL), is essentially an extended version of MacStarLogo with many additional features and capabilities.

Since then two multi−platform Java−based multi−agent Logos have been developed: NetLogo (from the CCL) and a Java−based version of StarLogo (from MIT).

The NetLogo language and environment differ in many respects from MIT StarLogo's. Both languages were inspired by the original StarLogo, but were redesigned in different ways. NetLogo's design was driven by the need to revise and expand the language so it is easier to use and more powerful, and by the need to support the HubNet architecture. NetLogo also incorporates almost all of the extended functionality of our earlier StarLogoT. See below for information on how to convert a StarLogoT model to NetLogo.

The "What is NetLogo?" section of this manual lists some of the other features that are new in NetLogo.

Downloading

The download form doesn't work for me. Can I have a direct link to the software?

Please write us at bugs@ccl.northwestern.edu and we'll work with you to either fix the problem with the download form, or provide an alternate method of downloading the software.

Downloading NetLogo takes too long. Can I get it some other way, like on a CD?

At present, no. If this is a problem for you, contact us at feedback@ccl.northwestern.edu.

I downloaded and installed NetLogo but the Models Library has few or no models in it. What's going on?

So far, users reporting this problem all used the "without VM" download option for Windows. Uninstall NetLogo and try the "with VM" download instead.

Even if that fixes it, please contact us at bugs@ccl.northwestern.edu. We'd like to fix this in a future version, but to troubleshoot it we need help from users.

Usage

How do I change how many patches there are?

Select the Graphics Window by dragging a rectangle around it with the mouse. Click the "Edit" button in the Toolbar and type in new values for "Screen Edge X" and "Screen Edge Y". (You can also right−click [Windows] or control−click [Mac] on the Graphics Window to edit it, or select it then double−click.)

How big can my model be? How many turtles, patches, procedures, buttons, etc.?

NetLogo has no fixed limits on size.

By default, though, on non−Macintosh operating systems NetLogo ships with a 512 megabyte ceiling on how much total RAM your model can use. But you can raise that limit by editing this section of the "NetLogo.lax" file in the NetLogo folder:

```
# LAX.NL.JAVA.OPTION.JAVA.HEAP.SIZE.MAX
# ------------------------------------
# allow the heap to get huge
lax.nl.java.option.java.heap.size.max=536870912
```

We have tested NetLogo with models that use hundreds of megabytes of RAM and they work fine. We haven't tested models that use gigabytes of RAM, though. It should work, but eventually perhaps you might hit some limits that are inherent in the underlying Java VM virtual machine and/or operating system (either built−in limits, or just bugs related to extremely high memory use).

When will NetLogo be a native app on Mac OS X?

We can't promise, but our plan is for NetLogo 1.2 to run natively under OS X. (The release of 1.2 is planned for Fall 2002.)

You can actually run NetLogo 1.1 natively under OS X, though this is not officially supported. If you want to try it, just double click the "nLogo.jar" file in the NetLogo folder. Note: you may encounter bugs and your model may run slowly. Note that it's only graphics that are slow, particularly turtle shapes. The `no−display` command will make your model run at normal speed, although you won't be able to see what's going on.

Can I import a graphic into NetLogo?

At present, no. We plan to add this capability in a future version.

My model runs slowly. How can I speed it up?

Here's some ways to make it run faster without changing the code:

- If your model is using all available RAM on your computer, then installing more RAM should help. If your hard drive makes a lot of noise while your model is running, you probably need more RAM.
- If you are on Windows, make sure you are running with the IBM 1.1 VM. (You can check in the "About NetLogo" window.) This is the VM that is bundled with NetLogo if you choose the "With VM" download option. NetLogo's graphics are much faster with this VM than any other VM. If you are not using the IBM 1.1 VM, then the next tip will be especially helpful:
- Use the `no-display` primitive to turn graphics off temporarily.

In many cases, though, if you want your model to run faster, you may need to make some changes to the code. Usually the most obvious opportunity for speedup is that you're doing too many computations that involve all the turtles or all the patches. Often this can be reduced by reworking the model so that it does less computation per time step. If you need help with this, if you contact us at feedback@ccl.northwestern.edu we may be able to help if you can send us your model or give us some idea of how it works. The members of the NetLogo Users Group may be able to help as well.

Improving the speed of NetLogo is a very high priority for us. You can expect future versions to be faster.

I want to try HubNet. Can I?

Currently using HubNet requires Texas Instruments' calculator network system, called Navigator. We plan to lift this requirement in the near future. In the meantime, Navigator is currently only available to select pilot sites, but TI has announced that it will be commercially available in January 2003.

Can I run a NetLogo model from the command line? Can I run it without a GUI?

At present there is no way to run without the GUI. We plan to add this in a future version.

Automating runs from the command line is possible, but currently not officially supported or documented. If you need this for your work, it can be done, but some light Java programming is required. We can supply sample Java code that you can alter for your needs. Anyone interested in doing this, write us at feedback@ccl.northwestern.edu.

Programming

How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?

We don't have a document that specifically summarizes the differences. We suggest reading the Programming Guide section of this manual (particularly the sections on "Ask" and "Agentsets"). Looking at some of the sample models and code examples in the Models Library may help as well.

If you need any help converting your StarLogo or StarLogoT model to NetLogo, please feel free to get in touch with us at feedback@ccl.northwestern.edu. The NetLogo Users Group is also a good

resource for getting help from other users.

The NetLogo world is a torus, that is, the edges of the screen are connected to each other, so turtles and patches "wrap around". Can I use a different world topology: bounded, infinite plane, sphere, etc.?

Torus is the only topology directly supported by NetLogo, but you can often simulate a different topology without too much extra effort.

If you want the world to be a bounded rectangle, you may need to add some code to your model to enforce this. Often a helpful technique is to turn the edge patches a different color, so turtles can easily detect when they "hit" the edge. Also, there are "no−wrap" versions of the "distance" and "towards" primitives; these should help.

If you want your turtles to move over an infinite plane, you can simulate this by having the turtles keep track of their position on the infinite plane, then hide the turtle when it goes "out of bounds". The Random Walk 360 model in the Models Library shows you how to code this.

Simulating a spherical or other topology might be difficult; we haven't seen a model that does this. (If you have one, please send it in!)

Does NetLogo have a command like StarLogo's "grab" command?

We don't have such a primitive, although we plan to add one −− or perhaps several! In the meantime, though, you can use the `without−interruption` primitive (new in NetLogo 1.1) to arrange exclusive interaction between agents. For example:

```
turtles−own [mate]
to setup
  ask turtles [ set mate nobody ]
end
to find−mate  ;; turtle procedure
  locals [candidate]
  without−interruption
    [ if mate = nobody
        [ set candidate random−one−of other−turtles−here with [mate = nobody]
          if candidate != nobody
            [ set mate candidate
              set mate−of candidate (turtle who) ] ] ] ;; turtle who = me!
end
```

Using `without−interruption` ensures that while a turtle is choosing a mate, all other agents are "frozen". This makes it impossible for two turtles to choose the same mate.

I tried to put `−at` after the name of a variable, for example `variable−at −1 0`, but NetLogo won't let me. Why not?

This syntax was supported by StarLogoT and some beta versions of NetLogo, but was removed from NetLogo 1.0. Instead, for a patch variable write e.g. `pcolor−of patch−at −1 0`, and for a turtle variable write e.g. `color−of one−of turtles−at −1 0`.

I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?

That math operations sometimes produce slightly inaccurate results like these is a limitation of IEEE floating point arithmetic in general. (NetLogo uses Java to do math, and Java implements the IEEE standard.) But there are some cases where NetLogo should compensate for it and doesn't. We fixed a number of them in the betas leading up to 1.0, and a fix for one more appeared in 1.1 (namely, sliders will no longer occasionally produce such slightly−off numbers). But in general, you shouldn't expect floating point arithmetic to produce absolutely exact answers.

The "precision" primitive is handy for rounding off numbers for display purposes.

If you encounter a specific instance of this problem and are not sure whether it falls into the "blame NetLogo" category or the "blame floating point arithmetic in general" category, run it by us and we'll take a look.

Is there some simple way to multiply one list by another list or number without writing a procedure?

At present, no. We plan to add this capability in a future version.

Can I read data from a file?

At present, this is only possible through the "Import World" mechanism, but you can't read files containing arbitrary data. We plan to add this capability in a future version of NetLogo.

In the meantime, you may find the following workarounds useful:

1) Copy and paste the data into the Procedures window, store it in a variable, and then deal with it from there.

2) Use "Import World" to import the data. This will require modifying your data to be in the format that import−world expects. You can use "Export World" to produce an example of the format, and then use that as a template.

How does NetLogo decide when to switch from agent to agent when running code?

Turtles are scheduled for execution in ascending order by ID number. Patches are scheduled for execution by row: left to right within each row, and starting with the top row. In a future version of NetLogo, we plan to add an option for randomized scheduling.

Once scheduled, an agent's "turn" ends only once it performs an action that affects the state of the world, such as moving, or creating a turtle, or changing the value of an agent variable. (Setting a local variable doesn't count.)

To prolong an agent's "turn", use the `without−interruption` command.

In general, we suggest you write your NetLogo code so that it does not depend on a particular scheduling mechanism. We make no guarantees that the scheduling algorithm will remain the same in future versions.

Primitives Dictionary

Alphabetical: A B C D E F G H I J L M N O P R S T U V W X
Categories: Turtle Patch Agentset Color Control/Logic Display HubNet I/O List String Math Plotting
Special: Variables Keywords Constants

Categories of Primitives

This is an approximate grouping. Remember that a turtle−related primitive might still be called by patches or observers, and vice versa. To see which agent (turtles, patches, observer) can actually run each command, consult each individual entry in the dictionary.

Turtle−related

back (bk) *BREED*−at *BREED*−here clear−turtles (ct) create−*BREED* create−custom−*BREED* create−custom−turtles (cct) create−turtles (crt) die distance distance−nowrap distancexy distancexy−nowrap downhill downhill4 dx dy forward (fd) hatch hideturtle (ht) home inspect jump left (lt) myself no−label nobody −of other−turtles−here other−*BREED*−here patch−here pen−down (pd) pen−up (pu) right (rt) set−default−shape setxy shape showturtle (st) sprout stamp towards towards−nowrap towardsxy towardsxy−nowrap turtle turtles turtles−at turtles−here uphill value−from

Patch−related primitives

clear−patches (cp) diffuse diffuse4 distance distance−nowrap distancexy distancexy−nowrap inspect myself neighbors neighbors4 no−label nobody nsum nsum4 −of patch patch−at patch−here patches sprout value−from

Agentset primitives

any ask at−points *BREED*−at *BREED*−here count histogram in−radius max−one−of min−one−of neighbors neighbors4 one−of other−turtles−here other−*BREED*−here patches random−one−of turtles with turtles−at turtles−here values−from

Color primitives

extract−hsb extract−rgb hsb rgb scale−color shade−of? wrap−color

Control flow and logic primitives

and end if ifelse locals loop not or repeat report ; (semicolon) stop startup timer to to−report wait while without−interruption xor

Display primitives

clear−all (ca) clear−graphics (cg) clear−patches (cp) clear−turtles (ct) display no−display no−label screen−edge−x screen−edge−y screen−size−x screen−size−y

HubNet primitives

hubnet−broadcast hubnet−fetch−message hubnet−message hubnet−message−source hubnet−message−tag hubnet−message−waiting? hubnet−reset hubnet−set−client−interface hubnet−set−tags

---

Input/output primitives

clear−output export−output export−plot export−world import−world print read−from−string show type user−choice user−input user−message

---

List primitives

but−first but−last empty? first fput is−list? item last length list lput member? position remove remove−duplicates replace−item reverse sentence sort values−from

---

String primitives

Operators (+, <, >, =, !=, <=, >=) but−first but−last empty? first is−string? item last length member? position remove read−from−string replace−item reverse substring word

---

Mathematical primitives

Arithmetic Operators (+, *, −, /, ^, <, >, =, !=, <=, >=) abs atan ceiling cos e exp floor int ln log max mean median min mod pi precision random random−seed round sin sqrt standard−deviation sum tan variance

---

Plotting primitives

autoplot? auto−plot−off auto−plot−on clear−all−plots clear−plot create−temporary−plot−pen export−plot histogram histogram−list plot plot−name plot−pen−down (ppd) plot−pen−reset plot−pen−up (ppu) plot−x−max plot−x−min plot−y−max plot−y−min plotxy ppd ppu set−current−plot set−current−plot−pen set−histogram−num−bars set−plot−pen−color set−plot−pen−interval set−plot−pen−mode set−plot−x−range set−plot−y−range

Predefined Variables

Turtles

breed
color
heading
hidden?
label
label−color
pen−down?
shape

size
who (read−only)
xcor
ycor

Patches

pcolor
plabel
plabel−color
pxcor
pycor

Keywords

breeds
end
globals
locals
patches−own
to
to−report
turtles−own

Constants

Mathematical Constants

e

2.718281828459045

pi

3.141592653589793

Boolean Constants

false
true

Color Constants

The allowable range of values for colors is 0 up to but not including 140. Each color ranges from black to white over a scale of ten. Thus the color red goes from black (10) to dark red (11) to red (15) to light red (19) to white (19.9). The scale is discontinuous; 19.9 is white, but 20.0 is black.

The available color names are listed below. (See also the rgb and hsb primitives.)

black = 0
gray = 5
white = 9.999
red = 15
orange = 25
brown = 35
yellow = 45
green = 55
lime = 65
turquoise = 75
cyan = 85
sky = 95
blue = 105
violet = 115
magenta = 125
pink = 135

A

abs

abs *number*

Reports the absolute value of *number*.

```
show abs -7
=> 7
show abs 5
=> 5
```

and

condition1 and *condition2*

Reports true if both *condition1* and *condition2* are true.

Note that if *condition1* is false, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ]  ;; the upper-right quadrant of
                       ;; patches turn blue
```

any

any *agentset*

Reports true if the given agentset is non-empty, false otherwise.

Equivalent to "*agentset* > 0", but more readable, and may run faster as well.

```
if any turtles with [color = red]
```

```
  [ show "at least one turtle is red!" ]
```

Arithmetic Operators (+, *, −, /, ^, <, >, =, !=, <=, >=)

All of these operators take two inputs, and all act as "infix operators" (going between the two inputs, as in standard mathematical use). NetLogo correctly supports order of operations for infix operators.

The operators work as follows: + is addition, * is multiplication, − is subtraction, / is division, ^ is exponentiation, < is less than, > is greater than, = is equal to, != is not equal to, <= is less than or equal, >= is greater than or equal.

All of the comparison operators also work on strings, and the addition operator (+) also functions as a string concatenation operator (see example below).

If you are not sure how NetLogo will interpret your code, you should insert parentheses.

```
show 5 * 6 + 6 / 3
=> 32
show 5 * (6 + 6) / 3
=> 20
show "tur" + "tle"
=> turtle
```

ask

ask *agentset* [*commands*]
ask *agent* [*commands*]

Takes a list of commands that will be run by the specified agent or agentset.

```
ask turtles [ fd 1 ]
  ;; all turtles move forward one step
ask patches [ set pcolor red ]
  ;; all patches turn red
ask turtle 4 [ rt 90 ]
  ;; only the turtle with id 4 turns right
```

at−points

***agentset* at−points [[*x1 y1*] [*x2 y2*] ...]**

Reduces the given agentset to include only the agents on the patches the given distances away from the calling agent. The distances are specified as a list of two−item lists, where the two items are the x and y offsets.

If the caller is the observer, then the points are measured relative to the origin, in other words, the points are taken as absolute patch coordinates.

If the caller is a turtle, the points are measured relative to the turtle's exact location, and not from the center of the patch under the turtle.

```
ask turtles at−points [[2 4] [1 2] [10 15]]
[ fd 1 ]  ;; only the turtles on the patches at the
```

```
;; distances (2,4), (1,2) and (10,15),
;; relative to the caller, move
```

atan

atan *x y*

Reports the arc tangent, in degrees (from 0 to 360), of x divided by y.

When y is 0: if x is positive, it reports 90; if x is zero, it reports 0; if x is negative, it reports 270.

Note that this version of atan is designed to conform to the geometry of the NetLogo world, where a heading of 0 is straight up, 90 is to the right, and so on clockwise around the circle. (Normally in geometry an angle of 0 is right, 90 is up, and so on, counterclockwise around the circle, and atan would be defined accordingly.)

```
show atan 1 −1
=> 135.0
show atan −1 1
=> 315.0
```

autoplot?

autoplot?

Reports true if auto−plotting is on for the current plot, false otherwise.

auto−plot−off
auto−plot−off

auto−plot−off
auto−plot−on

This pair of commands is used to control the NetLogo feature of auto−plotting in the current plot. Auto−plotting will automatically update the x and y axes of the plot whenever the current pen exceeds these boundaries. It is useful when wanting to display all plotted values in the current plot, regardless of the current plot ranges.

B

back
bk

back *number*

The turtle moves backward by *number* steps. (If *number* is negative, the turtle moves forward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `bk 0.5` and `bk`

1 both take one unit of time, but `bk 3` takes three.

See also forward, jump.

---

breeds

breeds [*breed1 breed2* ...]

This keyword, like the globals, turtles−own, and patches−own keywords, can only be used at the beginning of a program, before any function definitions. It defines breeds and their associated agentsets.

Any turtle of the given breed:

- is part of the agentset named by the breed name
- has its breed built−in variable set to that agentset

Most often, the agentset is used in conjunction with ask to give commands to only the turtles of a particular breed.

```
breeds [ mice frogs ]
to setup
  ca
  create−mice 50
  ask mice [ set color white ]
  create−frogs 50
  ask frogs [ set color green ]
  show breed−of one−of mice    ;; prints mice
  show breed−of one−of frogs   ;; prints frogs
end
```

See also globals, patches−own, turtles−own, *<BREED>*−own, create−*<BREED>*, create−custom−*<BREED>*, *<BREED>*−at, *<BREED>*−here.

---

but−first
bf
but−last
bl

but−first *list*
but−first *string*
but−last *list*
but−last *string*

When used on a list, but−first reports all of the list items of *list* except the first, and but−last reports all of the list items of *list* except the last.

On strings, but−first and but−last report a shorter string omitting the first or last character of the original string.

```
;; mylist is [2 4 6 5 8 12]
```

```
set mylist but-first mylist
;; mylist is now [4 6 5 8 12]
set mylist but-last mylist
;; mylist is now [4 6 8]
show but-first "string"
;; prints "tring"
show but-last "string"
;; prints "strin"
```

C

ceiling

ceiling *number*

Reports the smallest integer greater than or equal to *number*.

```
show ceiling 4.5
=> 5
show ceiling -4.5
=> -4
```

clear-all
ca

clear-all

Kills all turtles, resets all global variables to zero, and calls clear-patches and clear-all-plots.

clear-all-plots

clear-all-plots

Clears every plot in the model. See clear-plot for more information.

clear-graphics
cg

clear-graphics

Kills all turtles and clears all patches. Combines the effect of clear-turtles and clear-patches.

clear-output
cc

clear−output


Clears all text from the output portion of the Command Center.

---

clear−patches
cp

clear−patches


Clears the patches by resetting all patch variables to their default initial values, including setting their color to black.

---

clear−plot

clear−plot

In the current plot only, resets all plot pens, deletes all temporary plot pens, resets the plot to its default values (for x range, y range, etc.), and resets all permanent plot pens to their default values. The default values for the plot and for the permanent plot pens are set in the plot Edit dialog, which is displayed when you edit the plot. If there are no plot pens after deleting all temporary pens, that is to say if there are no permanent plot pens, a default plot pen will be created with the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Name: "default"
- Interval: 1.0

See also clear−all−plots.

---

clear−turtles
ct

clear−turtles


Kills all turtles.

See also die.

---

cos

cos *number*

Reports the cosine of the given angle. Assumes the angle is given in degrees.

```
show cos 180
=> −1.0
```

---

count

count *agentset*

Reports the number of agents in the given agentset.

```
show count turtles
;; prints the total number of turtles
show count patches with [pcolor = red]
;; prints the total number of red patches
```

---

create−turtles
crt
create−<*BREED*>

create−turtles *number*
create−<*BREED*> *number*


Creates *number* new turtles . New turtles start at position (0, 0), are created with the 14 primary colors, and have headings from 0 to 360, evenly spaced.

```
crt 100
ask turtles [ fd 10 ] ;; makes an evenly spaced circle
```

If the create−<*BREED*> form is used, the new turtles are created as members of the given breed.

---

create−custom−turtles
cct
create−custom−<*BREED*>
cct−<*BREED*>

create−custom−turtles *number* [*commands*]
create−custom−<*BREED*> *number* [*commands*]


Creates *number* new turtles (of the given breed, if specified). New turtles start at position (0, 0). New turtles are created with the 14 primary colors and have headings from 0 to 360, evenly spaced.

The new turtles immediately run *commands*. This is useful for giving the new turtles a different color, heading, or whatever.

```
breeds [canaries snakes]
```

```
to setup
  ca
  create-custom-canaries 50
    [ set color yellow ]
  create-custom-snakes 50
    [ set color green ]
end
```

Note: While the commands are running, no other agents are allowed to run any code (as with the without-interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen until they are fully initialized.

create-temporary-plot-pen

create-temporary-plot-pen *string*

A new temporary plot pen with the given name is created in the current plot and set to be the current pen.

Few models will want to use this primitive, because all temporary pens disappear when clear-plot or clear-all-plots are called. The normal way to make a pen is to make a permanent pen in the plot's Edit dialog.

If a temporary pen with that name already exists in the current plot, no new pen is created, and the existing pen is set to the the current pen. If a permanent pen with that name already exists in the current plot, a runtime error is thrown.

The new temporary plot pen has the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Interval: 1.0

Note that the old primitive set-plot-pen, which was used in earlier versions of NetLogo, is now just an alternate name for this primitive. set-plot-pen is available only for reasons of compatibility; new models should not use set-plot-pen.

See: clear-plot, clear-all-plots, and set-current-plot-pen.

D

die

die

The turtle dies.

```
if xcor > 20 [ die ]
```

```
;; all turtles with xcor greater than 20 die
```

See also: ct

diffuse

diffuse *patch−variable number*

Tells each patch to share (*number* * 100) percent of the value of *patch−variable* with its eight neighboring patches. *number* should be between 0 and 1.

Note that this is an observer command only, even though you might expect it to be a patch command. (The reason is that it acts on all the patches at once −− patch commands act on individual patches.)

```
diffuse chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the chemical
;; from each neighboring patch.)
```

diffuse4

diffuse4 *patch−variable number*

Like diffuse, but only diffuses to the four neighboring patches (to the north, south, east, and west), not to the diagonal neighbors.

```
diffuse4 chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 4 patches. Thus,
;; each patch gets 1/4 of 50% of the chemical
;; from each neighboring patch.)
```

display

display

Resumes updating of the graphics window (which may have been turned off using the no−display command). Any changes in the state of the graphics window that were made while no−display was in effect will immediately be drawn on−screen all at once.

See also no−display.

distance

distance *agent*

Reports the distance to the given turtle or patch.

Unlike "distance−nowrap", turtles and patches use the wrapped distance (around the edges of the screen) if that distance is shorter than the on−screen distance.

---

distance−nowrap

distance−nowrap *agent*

Reports the distance to the given turtle or patch.

Unlike "distance", this always reports the on−screen distance, never a distance that would require wrapping around the edges of the screen.

---

distancexy

distancexy *xcor ycor*

Reports the distance to the point (*xcor*, *ycor*)

Unlike "distancexy−nowrap", the wrapped distance (around the edges of the screen) is used if that distance is shorter than the on−screen distance.

```
if (distancexy 0 0) > 10
  [ set color green ]
;; all turtles more than 10 units from
;; the center of the screen turn green.
```

---

distancexy−nowrap

distancexy−nowrap *xcor ycor*

Reports the distance to the point (*xcor*, *ycor*)

Unlike "distancexy", this always reports the on−screen distance, never a distance that would require wrapping around the edges of the screen.

---

downhill

downhill *patch−variable*

Reports the turtle heading (between 0 and 359 degrees) in the direction of the minimum value of the variable *patch−variable*, of the patches in a one−patch radius of the turtle. (This could be as many as eight or as few as five patches, depending on the position of the turtle within its patch.)

If there are multiple patches that have the same smallest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need to reach the nearest corner of that patch is reported.

See also downhill4, uphill, uphill4.

---

downhill4

downhill4 *patch−variable*

Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the minimum value of the variable *patch−variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same least value, a random patch from those patches will be selected.

See also downhill, uphill, uphill4.

---

dx
dy

dx
dy

Reports the x−increment or y−increment if the turtle were to take one step forward in its current heading.

These primitives are useful for "testing" the patch ahead of the turtle before moving.

```
if (food-of patch-at dx dy) > 0
  [ fd 1
    eat ]
```

E

---

empty?

empty? *list*
empty? *string*

Reports true if the given list or string is empty, false otherwise.

Note: the empty list is written `[]`. The empty string is written `""`.

---

end

end

Used to conclude a procedure. See to and to−report.

---

every

every *number* [*commands*]

Runs the given commands at most every *number* seconds.

More technically, the behavior of every can described as follows. When an agent reaches an "every", it checks a timer to see if the given amount of time has passed since the last time the same agent ran the commands in the "every" in the same context. If so, it runs the commands; otherwise they are skipped and execution continues. (The next time the agent reaches the "every", the commands will get another chance to run.)

Here, "in the same context" means during the same ask (or button press or command typed in the Command Center). So it doesn't make sense to write `ask turtles [ every 0.5 [ ... ] ]`, because when the ask finishes the turtles will all discard their timers for the "every". The correct usage is shown below.

```
every 0.5 [ ask turtles [ fd 1 ] ]
;; twice a second the turtles will move forward 1
every 2 [ set index index + 1 ]
;; every 2 seconds index is incremented
```

See also wait.

---

exp

exp *number*

Reports the value of e raised to the *number* power.

Note: This is the same as e ^ *number*.

---

export−output
export−plot
export−world

export−output *filename*
export−plot *plotname filename*
export−world *filename*

export−output writes the contents of the output portion of the Command Center to an external file given by the string *filename*.

export−plot writes the x and y values of all points plotted by all the plot pens in the plot given by the string *plotname* to an external file given by the string *filename*. If a pen is in bar mode (mode 0) and the y value of the point plotted is greater than 0, the upper−left corner point of the bar will be exported. If the y value is less than 0, then the lower−left corner point of the bar will be exported.

export−world writes the values of all variables, both built−in and user−defined, including all observer, turtle, and patch variables, to an external file given by the string *filename*. (The result file can be read back into NetLogo with the import−world primitive.)

export−plot and export−world save files in in plain−text, "comma−separated values" (`.csv`) format. CSV files can be read by most popular spreadsheet and database programs as well as any text editor.

If the file already exists, it is overwritten.

If you wish to export to a file in a location other than the NetLogo folder, you should include the full path to the file you wish to export. (Use the forward−slash "/" as the folder separator.)

Note that the functionality of these primitives is also available directly from NetLogo's File menu.

```
export−world "fire.csv"
;; exports the state of the model to the file fire.csv
;; located in the NetLogo folder
export−plot "Temperature" "c:/My Documents/plot.csv"
;; exports the plot named
;; "Temperature" to the file plot.csv located in
;; the C:\My Documents folder
```

---

extract−hsb

extract−hsb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the hue, saturation and brightness, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract−hsb red
=> [0.0 1.0 1.0]
show extract−hsb cyan
=> [0.5 1.0 1.0]
```

See also hsb, rgb, extract−rgb.

extract−rgb

extract−rgb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the levels of red, green, and blue, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract−rgb red
=> [1.0 0.0 0.0]
show extract−rgb cyan
=> [0.0 1.0 1.0]
```

See also rgb, hsb, extract−hsb.

F

first

first *list*
first *string*

On a list, reports the first (0th) item in the list.

On a string, reports a one−character string containing only the first character of the original string.

floor

floor *number*

Reports the largest integer less than or equal to *number*.

```
show floor 4.5
=> 4
show floor −4.5
=> −5
```

forward
fd

forward *number*

The turtle moves forward by *number* steps. (If *number* is negative, the turtle moves backward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `fd 0.5` and `fd 1` both take one unit of time, but `fd 3` takes three.

See also jump.

fput

fput *item list*

Adds *item* to the beginning of a list and reports the new list.

```
;; suppose mylist is [5 7 10]
set mylist fput 2 mylist
;; mylist is now [2 5 7 10]
```

G

globals

globals [*var1 var2* ...]

This keyword, like the breeds, <*BREED*>−own, patches−own, and turtles−own keywords, can only be used at the beginning of a program, before any function definitions. It defines new global variables. Global variables are "global" because they are accessible by all agents and can be used anywhere in a model.

Most often, globals is used to define variables or constants that need to be used in many parts of the program.

H

hatch

hatch *number* [*commands*]

Each turtle creates *number* new turtles, each identical to itself, and tells the new turtles to run *commands*. This is useful for giving the new turtles different colors, headings, breeds, or whatever.

Note: While the commands are running, no other agents are allowed to run any code (as with the without−interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on−screen in an only partly initialized state.

```
hatch 1 [ lt 45 fd 1 ]
;; each turtle creates one new turtle,
;; and the child turns and moves away
```

hideturtle
ht

hideturtle


The turtle makes itself invisible.

Note: This command is equivalent to setting the turtle variable "hidden?" to true.

See also showturtle.

---

histogram

histogram *agentset* [*reporter*]

Resets the current plot pen, then draws a histogram of the values reported when all agents in the agentset run the given reporter. (It should report a numeric value. Any non−numeric values reported are ignored.)

The histogram is drawn on the current plot using the current plot pen and pen color. Use set−plot−x−range to control the range of values to be histogrammed, and set the pen interval (either directly with set−plot−pen−interval, or indirectly via set−histogram−num−bars) to control how many bars that range is split up into.

Be sure that if you want the histogram drawn with bars that the current pen is in bar mode (mode 1).

```
histogram turtles [color]
;; draws a histogram showing how many turtles there are
;; of each color
```

Note: using this primitive amounts to the same thing as writing: `histogram−list values−from` *`agentset`* `[` *`reporter`* `]`, but is more efficient.

---

histogram−list

histogram−list *list*

Resets the current plot pen, then draws a histogram of the values in the given list.

See histogram, above, for more information.

---

home

home


Returns turtle to the origin. Equivalent to `setxy 0 0`.

---

hsb

hsb *hue saturation brightness*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the HSB spectrum, in NetLogo's color space.

All three values should be in the range 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (It contains only certain discrete hues, and for each hue, either saturation or brightness may vary, but not both −− at least one of the two is always 1.0.)

```
show hsb 0 0 0
=> 0  ;; (black)
show hsb 0.5 1.0 1.0
=> 85 ;; (cyan)
```

See also extract−hsb, rgb, extract−rgb.

---

hubnet−broadcast

hubnet−broadcast *variable−name value*

This broadcasts *value* from NetLogo to the variable *variable−name* on the teacher calculator. You may send a number, a string, a list of numbers, or a matrix (a list of lists) of numbers.

See the HubNet Programming Guide for details and instructions.

---

hubnet−fetch−message

hubnet−fetch−message

This retrieves a new message from the server if there is one to be retrieved.

See the HubNet Programming Guide for details.

---

hubnet−message

hubnet−message

Reports the message retrieved by hubnet−fetch−message.

See the HubNet Programming Guide for details.

---

hubnet−message−source

hubnet−message−source

Reports the id of the device that sent the message retrieved by hubnet−fetch−message.

See the HubNet Programming Guide for details.

---

hubnet−message−tag

hubnet−message−tag

Reports the variable name on the device that sent the message that was retrieved by hubnet−fetch−message.

See the HubNet Programming Guide for details.

---

hubnet−message−waiting?

hubnet−message−waiting?

This looks for a new message on the server, and reports true if it found one, false otherwise.

See the HubNet Programming Guide for details.

---

hubnet−reset

hubnet−reset

Logs you into the HubNet system. You must be logged in to use any of the other hubnet−primitives.

See the HubNet Programming Guide for details.

---

hubnet−set−client−interface

hubnet−set−client−interface *client−type activity−name*

If *client−type* is "TI−83+", notify the user to enable the activity with the given name on the TI Navigator web site.

Future versions of HubNet may support other client types, and/or change the meaning of the second input to this command.

See the HubNet Programming Guide for details.

---

hubnet−set−tags

hubnet−set−tags *variable−list*

This sets which variables NetLogo expects from the calculators. NetLogo will only check for these variables and will ignore all others.

```
hubnet-set-tags ["L1" "LOCS"]
;; looks for the calculator lists L1 and LOCS on the server
```

See the HubNet Programming Guide for details.

I

if

if *condition* [*commands*]

If *condition* reports true, runs *commands*.

```
if (xcor > 0) [ set color blue ]
;; turtles on the right half of the screen
;; turn blue
```

ifelse

ifelse *reporter* [*commands1*] [*commands2*]

Reporter must report a boolean (true or false) value.

If *reporter* reports true, runs *commands1*.

If *reporter* reports false, runs *commands2*.

The reporter may report a different value for different agents, so some agents may run *commands1* while others run *commands2*.

```
ask patches
  [ ifelse (pxcor > 0)
      [ set pcolor blue ]
      [ set pcolor red ] ]
;; the left half of the screen turns red and
;; the right half turns blue
```

import-world

import-world *filename*

Reads the values of all variables for a model, both built-in and user-defined, including all observer, turtle, and patch variables, from an external file named by the given string. The file should be in the format used by the export-world primitive.

Note that the functionality of this primitive is also directly available from NetLogo's File menu.

When using import-world, to avoid errors, perform these steps in the following order:

1. Open the model from which you created the export file.
2. Press the Setup button, to get the model in a state from which it can be run.

3. Import the file.
4. If you want, press Go button to continue running the model from the point where it left off.

If you wish to import a file from a location other than the NetLogo folder, you may include the full path to the file you wish to import. See export−world for an example.

in−radius

agentset* in−radius *number

Reports an agentset that includes only those agents from the original agentset whose distance from the caller is less than or equal to *number*.

```
ask turtles
  [ ask patches in−radius 3
      [ set pcolor red ] ]
;; each turtle makes a red "splotch" around itself
```

inspect

inspect *agent*

Opens an agent monitor for the given agent (turtle or patch).

```
inspect patch 2 4
;; an agent monitor opens for that patch
inspect random−one−of sheep
;; an agent monitor opens for a random turtle from
;; the "sheep" breed
```

int

int *number*

Reports the integer part of number −− any fractional part is discarded.

```
show int 4.7
=> 4
show int −3.5
=> 3
```

is−list?

is−list? *value*

Reports true if *value* is a list, false otherwise.

item

item *index list*
item *index string*

On lists, reports the value of the item in the given list with the given index.

On strings, reports the character in the given string at the given index.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my−shoe"
=> "s"
```

J

jump

jump *number*


Turtles move forward by *number* units all at once, without the amount of time passing depending on the distance.

This command is useful for synchronizing turtle movements. The command forward 15 takes 15 times longer to run than forward 1, but jump 15 runs in the same amount of time as forward 1.

Note: When turtles jump, they do not step on any of the patches along their path.

L

last

last *list*
last *string*

On a list, reports the last item in the list.

On a string, reports a one−character string containing only the last character of the original string.

left
lt

left *number*

The turtle turns left by *number* degrees. (If *number* is negative, it turns right.)

---

length

length *list*
length *string*

Reports the number of elements in the given list, or the number of characters in the given string.

---

list

list *value1 value2*

Reports a two−item list containing the given items. The items can be of any type, produced by any kind of reporter.

```
show list (random 10) (random 10)
=> [4 9]  ;; or similar list
```

---

ln

ln *number*

Reports the natural logarithm of *number*, that is, the logarithm to the base e (2.71828...).

See also e, log.

---

locals

locals [*var1 var2* ...]

Locals is a keyword used to declare "local" variables in a procedure, that is, variables that are usable only within that procedure. It must appear at the beginning of the procedure, before any commands.

A local variable may not have the same name as an existing observer, turtle, or patch variable.

See to and to−report.

```
to hunt  ;; turtle procedure
  locals [prey]
  set prey random-one-of other-turtles-here
  if prey != nobody
    [ eat prey ]
end
```

---

log

log *number base*

Reports the logarithm of *number* in base *base*.

```
show log 64 2
=> 6
```

See also ln.

---

loop

loop [*commands*]

Runs the list of commands forever, or until the current procedure exits through use of the stop command or the report command.

Note: In most circumstances, you should use a forever button in order to repeat something forever. The advantage of using a forever button is that the user can click the button to stop the loop.

---

lput

lput *value list*

Adds *value* to the end of a list and reports the new list.

```
;; suppose mylist is [2 7 10 "Bob"]
set mylist lput 42 mylist
;; mylist now is [2 7 10 "Bob" 42]
```

M

---

max

max *list*

Reports the maximum number value in the list. It ignores other types of items.

```
show max values-from turtles [xcor]
;; prints the x coordinate of the turtle which is
;; farthest right on the screen
```

---

max-one-of

max-one-of *agentset* [*reporter*]

Reports the agent in the agentset that has the highest value for the given reporter.

```
show max-one-of patches [count turtles-here]
```

```
;; prints the patch with the most turtles on it
```

mean

mean *list*

Reports the statistical mean of the numeric elements of the given list. Ignores non−numeric elements. The mean is the average, i.e., the sum of the elements divided by the total number of elements.

```
show mean values−from turtles [xcor]
;; prints the average of all the turtles' x coordinates
```

median

median *list*

Reports the statistical median of the numeric elements of the given list. Ignores non−numeric elements. The median is the item that would be in the middle if all the items were arranged in order. (If two items would be in the middle, the median is the average of the two.)

```
show median values−from turtles [xcor]
;; prints the median of all the turtles' x coordinates
```

member?

member? *value list*
member? *string1 string2*

For a list, reports true if the given value appears in the given list, otherwise reports false.

For a string, reports true or false depending on whether *string1* appears anywhere inside *string2* as a substring.

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "rin" "string"
=> true
```

See also position.

min

min *list*

Reports the minimum number value in the list. It ignores other types of items.

```
show min values−from turtles [xcor]
;; prints the lowest x−coordinate of all the turtles
```

min−one−of

min−one−of *agentset* [*reporter*]

Reports the agent in the agentset that reports the lowest value for the given reporter.

```
show min−one−of turtles [xcor + ycor]
;; reports the turtle with the smallest sum of
;; its coordinates
```

mod

number1* mod *number2

Reports *number1* modulo *number2*: that is, the remainder when *number1* is divided by *number2*.

Note that mod is "infix", that is, it comes between its two inputs.

```
show 62 mod 5
=> 2
```

mouse−down?

mouse−down?

Reports true if the mouse button is down, false otherwise.

Note: If the mouse pointer is outside of the NetLogo graphics window, mouse−down? will always report false.

mouse−xcor
mouse−ycor

mouse−xcor
mouse−ycor

Reports the x or y coordinate of the mouse in the Graphics Window. The value is in terms of turtle coordinates, so it is a floating−point number. If you want patch coordinates, use `round mouse−xcor` and `round mouse−ycor`.

Note: If the mouse is outside of the NetLogo graphics window, reports the value from the last time it was inside.

```
;; to make the mouse "draw" in red:
if mouse−down?
  [ set pcolor−of patch−at mouse−xcor mouse−ycor red ]
```

myself

myself


When an agent has been asked by a turtle to run some code, using myself in that code reports the agent (turtle or patch) that did the asking.

myself is most often used in conjunction with −of to read or set variables in the asking agent.

myself can be used within blocks of code not just in the ask command, but also hatch, values−from, value−from, with, min−one−of, max−one−of, and histogram.

```
ask turtles
  [ ask patches in-radius 3
      [ set pcolor color-of myself ] ]
;; each turtle makes a colored "splotch" around itself
```

See the "Myself Example" code example for more examples.

N

neighbors
neighbors4

neighbors
neighbors4


Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4).

```
show sum values-from neighbors [count turtles-here]
  ;; prints the total number of turtles on the eight
  ;; patches around the calling turtle or patch
ask neighbors4 [ set pcolor red ]
  ;; turns the four neighboring patches red
```

no−display

no−display


Turns off all updating of the graphics window until the display command is issued. This has two major uses. One, your model will run faster when graphics updating is off, so if you're in a hurry, this command will let you get results faster. Two, you can control when the user sees screen updates. You might want to change lots of things on the screen behind the user's back, so to speak, then make them visible to the user all at once.

See also display.

no−label

no−label

This is a special value used to remove labels from turtles and patches.

When you set a turtle's label to no−label, or a patch's plabel to no−label, then a label will no longer be drawn on top of the turtle or patch.

---

nobody

nobody

This is a special value which some primitives such as turtle, random−one−of, max−one−of, etc. return to indicate that no agent was found. Also, when a turtle dies, it becomes equal to nobody.

```
set other random-one-of other-turtles-here
if other != nobody
  [ set color-of other red ]
```

---

not

not *boolean*

Reports true if *boolean* is false, otherwise reports false.

```
if not (color = blue) [ fd 10 ]
;; all non-blue turtles move forward 10 steps
```

---

nsum
nsum4

nsum *patch−variable*
nsum4 *patch−variable*

For each patch, reports the sum of the values of *patch−variable* in the 8 surrounding patches (nsum) or 4 surrounding patches (nsum4).

Note that nsum/nsum4 are equivalent to the combination of the sum, values−from, and neighbors/neighbors4 primitives:

```
sum values-from neighbors [var]
  ;; does the same thing as "nsum var"
sum values-from neighbors4 [var]
  ;; does the same thing as "nsum4 var"
```

Therefore nsum and nsum4 are included as separate primitives mainly for backwards compatibility with older versions of NetLogo, which did not have the neighbors and neighbors4 primitives.

See also neighbors, neighbors4.

O

−of

VARIABLE*−of *agent

Reports the value of the *VARIABLE* of the given agent. Can also be used to set the value of the variable.

```
set color−of random−one−of turtles red
;; a randomly chosen turtle turns red
ask turtles [ set pcolor−of (patch−at −1 0) red ]
;; each turtle turns the patch on its left red
```

one−of

one−of *agentset*

If given a turtle agentset, reports the turtle in the set with the lowest numbered ID.

If given a patch agentset, reports the patch in the set with the highest pycor and, if a tie−breaker is needed, with the lowest pxcor.

If the agentset is empty, reports nobody.

See also random−one−of.

or

boolean1* or *boolean2

Reports true if either *boolean1* or *boolean2*, or both, is true.

Note that if *condition1* is true, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower−left quadrant
```

other−turtles−here
other−*BREED*−here

other−turtles−here
other−*BREED*−here

Reports an agentset consisting of all turtles on the calling turtle's patch (*not* including the caller itself). If a breed is specified, only turtles with the given breed are included.

```
;; suppose I am one of 10 turtles on the same patch
show count other−turtles−here
```

```
=> 9
```

Example using breeds:

```
breeds [cats dogs]
show count other-dogs-here
;; prints the number of dogs (that are not me) on my patch
```

See also turtles-here.

P

patch

patch *pxcor pycor*

Given two integers, reports the single patch with the given pxcor and pycor. (The coordinates are the actual coordinates; they are not computed relative to the calling agent, as with patch-at.) *pxcor* and *pycor* must be integers.

```
ask (patch 3 -4) [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 turns green
```

See also patch-at.

patch-at

patch-at *dx dy*

patch-at reports the single patch at (dx, dy) from the caller, that is, dx patches east and dy patches north of the caller. (If the caller is the observer, the given offsets are computed from the origin.)

```
ask patch-at 1 -1 [ set pcolor green ]
;; if caller is the observer, turn the patch
;;   at (1, -1) green
;; if caller is a turtle or patch, turns the
;;   patch just southeast of the caller green
```

See also patch.

patch-here

patch-here

patch-here reports the patch under the turtle.

patches

patches

Reports the agentset consisting of all patches.

---

patches−own

patches−own [*var1 var2* ...]

This keyword, like the globals, breeds, <*BREED*>−own, and turtles−own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables that all patches can use.

All patches will then have the given variables and be able to use them.

See also globals, turtles−own, breeds, <*BREED*>−own.

---

pen−down
pd
pen−up
pu

pen−down
pen−up

The turtle its pen down (or up), so that it draws (leaves a trail) when they move (or doesn't).

Turtles draw by changing the color of the patches underneath them to their own color. To change the color of the turtle's pen (and the color of the turtle itself), use `set color`.

Note: When a turtle's pen is down, only the commands forward and back cause drawing.

Note: Theses commands are equivalent to setting the turtle variable "pen−down?" to true or false.

---

plot

plot *number*

Increments the x−value of the plot pen by plot−pen−interval, then plots a point at the updated x−value and a y−value of *number*. (The first time the command is used on a plot, the point plotted has an x−value of 0.)

---

plot−name

plot−name

Reports the name of the current plot (a string).

---

plot−pen−down
ppd
plot−pen−up
ppu

plot−pen−down
plot−pen−up

Puts down (or up) the current plot−pen, so that it draws (or doesn't). (By default, all pens are down initially.)

---

plot−pen−reset

plot−pen−reset

Clears everything the current plot pen has drawn, moves it to (0,0), and puts it down. If the pen is a permanent pen, the color and mode are reset to the default values from the plot Edit dialog.

---

plotxy

plotxy *number1 number2*

Moves the current plot pen to the point with coordinates (*number1*, *number2*). If the pen is down, a line, bar, or point will be drawn (depending on the pen's mode).

---

plot−x−min
plot−x−max
plot−y−min
plot−y−max

plot−x−min
plot−x−max
plot−y−min
plot−y−max

Reports the minimum or maximum value on the x or y axis of the current plot.

These values can be set with the commands set−plot−x−range and set−plot−y−range. (Their default values are set from the plot Edit dialog.)

---

position

position *item list*
position *string1 string2*

On a list, reports the first position of *item* in *list*, or false if it does not appear.

On strings, reports the position of the first appearance *string1* as a substring of *string2*, or false if it does not appear.

Note: The positions are numbered beginning with 0, not with 1.

```
;; suppose mylist is [2 7 4 7 "Bob"]
show position 7 mylist
=> 1
show position 10 mylist
=> false
show position "rin" "string"
=> 2
```

See also member?.

---

precision

precision *number places*

Reports *number* rounded to *places* decimal places.

If *places* is negative, the rounding takes place to the left of the decimal point.

```
show precision 1.23456789 3
=> 1.235
show precision 3834 -3
=> 4000
```

---

print

print *value*

Prints *value* in the Command Center, followed by a carriage return.

The calling agent is *not* printed before the value, unlike show.

See also show, type.

R

---

random

random *number*

If *number* is positive, reports a random number greater than or equal to 0 but strictly less than *number*.

If *number* is negative, the number reported is less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always zero as well.

If *number* is an integer, reports a random integer.

If *number* is floating point (has a decimal point), reports a floating point number.

```
show random 3
;; prints 0, 1,  or 2
show random 5.0
;; prints a number at least 0.0 but less than 5.0,
;; for example 4.686596634174661
```

---

random−one−of

random−one−of *agentset*

Reports a random agent chosen from the agentset. If the agentset is empty, reports nobody.

```
ask random−one−of patches [ set pcolor green ]
;; a random patch turns green
set pcolor−of random−one−of patches green
;; another way to say the same thing
ask patches with [any turtles−here]
  [ show random−one−of turtles−here ]
;; for each patch containing turtles, prints one of
;; those turtles
```

See also one−of.

---

random−seed

random−seed *number*

Sets the seed of the pseudo−random number generator to the integer part of *number*. The exact value of the seed does not matter except as indicated below.

If the pseudo−random number generator is initialized to a fixed seed, then it will generate exactly the same random sequence every time a model is run.

On the other hand, setting the seed to different values in each run will make the sequences different for each run.

When this command is not used, the pseudo−random number generator uses a seed based on the current time.

```
random−seed 47823
show random 100
=> 57
show random 100
=> 91
random−seed 47823
show random 100
=> 57
show random 100
=> 91
```

read−from−string

read−from−string *string*

Interprets the given string as if it had been typed in the Command Center, and reports the resulting value. The result may be a number, list, string, or boolean value, or the special value "nobody".

Useful in conjunction with the user−input primitive for converting the user's input into usable form.

```
show read−from−string "3" + read−from−string "5"
=> 8
show length read−from−string "[1 2 3]"
=> 3
crt read−from−string user−input "Make how many turtles?"
;; the number of turtles input by the user
;; are created
```

remove

remove *item list*

For a list, reports a copy of *list* with all instances of *item* removed.

For strings, reports a copy of *string2* with all the appearances of *string1* as a substring removed.

```
set mylist [2 7 4 7 "Bob"]
set mylist remove 7 mylist
;; mylist is now [2 4 "Bob"]
show remove "na" "banana"
=> "ba"
```

remove−duplicates

remove−duplicates *list*

Reports a copy of *list* with all duplicate items removed. The first of each item remains in place.

```
set mylist [2 7 4 7 "Bob" 7]
set mylist remove−duplicates mylist
;; mylist is now [2 7 4 "Bob"]
```

repeat

repeat *number* [*commands*]

Runs *commands number* times.

```
pd repeat 36 [ fd 1 rt 10 ]
;; the turtle draws a circle
```

replace−item

replace−item *index list value*
replace−item *index string1 string2*

On a list, replaces an item in that list. *index* is the index of the item to be replaced, starting with 0. (The 6th item in a list would have an index of 5.) Note that "replace−item" is used in conjunction with "set" to change a list.

Likewise for a string, but the given character of *string1* removed and the contents of *string2* spliced in instead.

```
show replace−item 2 [2 7 4 5] 15
=> 2 7 15 5
show replace−item 1 "sat" "lo"
=> "slot"
```

report

report *value*

Immediately exits from the current to−report procedure and reports *value* as the result of that procedure. report and to−report are always used in conjunction with each other. See to−report for a discussion of how to use them.

reset−timer

reset−timer

Resets the global clock to zero. See also timer.

reverse

reverse *list*
reverse *string*

Reports a reversed copy of the given list or string.

```
show mylist
;; mylist is [2 7 4 "Bob"]
```

```
set mylist reverse mylist
;; mylist now is ["Bob" 4 7 2]
show reverse "string"
=> "gnirts"
```

rgb

rgb *red green blue*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the RGB spectrum, in NetLogo's color space.

All three inputs should be in the range 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (See hsb for a description of what parts of the HSB color space NetLogo colors cover; this is difficult to characterize in RGB terms.)

```
show rgb 0 0 0
=> 0  ;; black
show rgb 0 1.0 1.0
=> 85 ;; cyan
```

See also extract−rgb, hsb, and extract−hsb.

right
rt

right *number*

The turtle turns right by *number* degrees. (If *number* is negative, it turns left.)

round

round *number*

Reports the integer nearest to *number*.

If the decimal portion of *number* is exactly .5, the number is rounded in the **positive** direction.

Note that rounding in the positive direction is not always how rounding is done in other software programs. (In particular, it does not match the behavior of the StarLogoT, which always rounded numbers ending in 0.5 to the nearest even integer.) The rationale for this behavior is that it matches how turtle coordinates relate to patch coordinates in NetLogo. For example, if a turtle's xcor is −4.5, then it is on the boundary between a patch whose pxcor is −4 and a patch whose pxcor is −5, but the turtle must be considered to be in one patch or the other, so the turtle is considered to be in the patch whose pxcor is −4, because we round towards the positive numbers.

```
show round 4.2
=> 4
```

```
show round 4.5
=> 5
show round −4.5
=> −4
```

S

scale−color

scale−color *color number range1 range2*

Reports a shade of *color* proportional to *number*.

If *range1* is less than *range2*, then the larger the number, the lighter the shade of *color*. But if *range2* is less than *range1*, the color scaling is inverted.

If *number* is less than *range1*, then the darkest shade of *color* is chosen.

If *number* is greater than *range2*, then the lightest shade of *color* is chosen.

Note: for *color* shade is irrelevant, e.g. green and green + 2 are equivalent, and the same spectrum of colors will be used.

```
ask turtles [ set color scale−color red age 0 50 ]
;; colors each turtle a shade of red proportional
;; to its value for the age variable
```

screen−edge−x
screen−edge−y

screen−edge−x
screen−edge−y

These reporters give the maximum x−coordinate and maximum y−coordinate (respectively) of the Graphics Window.

screen−edge−x and −y are the "half−width" and "half−height" of the NetLogo world −− the distances from the origin to the edges. screen−size is the same as ((2 * screen−edge) + 1).

Note: You can set the size of the Graphics Window only by editing it −− these are reporters which cannot be set.

```
cct 100 [ setxy (random screen−edge−x)
                (random screen−edge−y) ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

screen−size−x
screen−size−y

screen−size−x
screen−size−y

These reporters give the total width and height of the NetLogo world.

Screen−size is the same as ((2 * screen−edge) + 1).

; (semicolon)

; *comments*

After a semicolon, the rest of the line is ignored. This is useful for adding "comments" to your code −− text that explains the code to human readers. Extra semicolons can be added for visual effect.

---

sentence
se

sentence *value1 value2*

Makes a list out of the values. If either value is a list, its elements are included in the result directly, rather than being included as a sublist. Examples make this clearer:

```
show sentence 1 2
=> [1 2]
show sentence [1 2] 3
=> [1 2 3]
show sentence 1 [2 3]
=> [1 2 3]
show sentence [1 2] [3 4]
=> [1 2 3 4]
```

---

set

set *variable value*

Sets *variable* to the given value.

Variable can be any of the following:

- An global variable declared using "globals"
- The global variable associated with a slider or switch
- A variable belonging to the calling agent
- If the calling agent is a turtle, a variable belonging to the patch under the turtle.
- An expression of the form *VARIABLE*−of *agent*

---

set−current−plot

set−current−plot *plotname*

Sets the current plot to the plot with the given name (a string). Subsequent plotting commands will affect the current plot.

---

set−current−plot−pen

set−current−plot−pen *penname*

The current plot's current pen is set to the pen named *penname* (a string). If no such pen exists in the current plot, a runtime error occurs.

---

set−default−shape

set−default−shape turtles *string*
set−default−shape *breed string*


Specifies a default initial shape for all turtles, or for a particular breed. When a turtle is created, or it changes breeds, it shape is set to the given shape.

The specified breed must be either be turtles or a breed defined by the breeds keyword, and the specified string must be the name of a currently defined shape.

In new models, the default shape for all turtles is "default".

Note that specifying a default shape does not prevent you from changing an individual turtle's shape later; turtles don't have to be stuck with their breed's default shape.

```
create−turtles 1 ;; new turtle's shape is "default"
create−cats 1    ;; new turtle's shape is "default"

set−default−shape turtles "circle"
create−turtles 1 ;; new turtle's shape is "circle"
create−cats 1    ;; new turtle's shape is "circle"

set−default−shape cats "cat"
set−default−shape dogs "dog"
create−cats 1   ;; new turtle's shape is "cat"
ask cats [ set breed dogs ]
  ;; all cats become dogs, and automatically
  ;; change their shape to "dog"
```

---

set−histogram−num−bars

set−histogram−num−bars *integer*

Set the current plot pen's plot interval so that, given the current x range for the plot, there would be *integer* number of bars drawn if the histogram or histogram−list commands were called.

See also histogram.

set−plot−pen−color

set−plot−pen−color *number*

Sets the color of the current plot pen to *number*.

set−plot−pen−interval

set−plot−pen−interval *number*

Tells the current plot pen to move a distance of *number* in the x direction during each use of the plot command. (The plot pen interval also affects the behavior of the histogram and histogram−list commands.)

set−plot−pen−mode

set−plot−pen−mode *number*

Sets the mode the current plot pen draws in to *number*. The allowed plot pen modes are:

- 0 (line mode) the plot pen draws a line connecting two points together.
- 1 (bar mode): the plot pen draws a bar of width plot−pen−interval with the point plotted as the upper (or lower, if you are plotting a negative number) left corner of the bar.
- 2 (point mode): the plot pen draws a point at the point plotted. Points are not connected.

The default mode for new pens is 0 (line mode).

set−plot−x−range
set−plot−y−range

set−plot−x−range *min max*
set−plot−y−range *min max*

Sets the minimum and maximum values of the x or y axis of the current plot.

The change is temporary and is not saved with the model. When the plot is cleared, the ranges will revert to their default values as set in the plot's Edit dialog.

setxy

setxy *x y*

The turtle sets its x−coordinate to *x* and its y−coordinate to *y*.

Equivalent to `set xcor x set ycor y`, except it happens in one time step instead of two.

```
setxy 0 0
;; turtle moves to the middle of the center patch
```

shade−of?

shade−of? ***color1 color2***

Reports true if both colors are shades of one another, false otherwise.

```
show shade−of? blue red
=> false
show shade−of? blue (blue + 1)
=> true
show shade−of? gray white
=> true
```

show

show ***value***

Prints *value* in the Command Center, preceded by the calling agent, and followed by a carriage return. (The calling agent is included to help you keep track of what agents are producing which lines of output.)

See also print, type.

showturtle
st

showturtle

The turtle becomes visible again.

Note: This command is equivalent to setting the turtle variable "hidden?" to false.

See also hideturtle.

sin

sin ***number***

Reports the sine of the given angle. Assumes angle is given in degrees.

```
show sin 270
=> −1.0
```

sort

sort *list*

Reports a new list containing the same elements as the input list, but in ascending order.

If there is at least one number in the list, the list is sorted in numerically ascending order and any non−numeric elements of the input list are discarded.

If there are no numbers, but at least one string in the list, the list is sorted in alphabetically ascending order and any non−string elements are discarded.

---

sprout

sprout *number* [*commands*]

Creates *number* new turtles on the current patch. The new turtles have random colors and orientations, and they immediately run *commands*. This is useful for giving the new turtles different colors, headings, breeds, or whatever.

```
sprout 1 [ set color red ]
```

Note: While the commands are running, no other agents are allowed to run any code (as with the without−interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on−screen until they are fully initialized.

---

sqrt

sqrt *number*

Reports the square root of *number*.

---

stamp

stamp *color*

Sets the color of the patch under the turtle to the given color.

```
repeat 30 [ stamp yellow fd 3 rt 6 ]
;; the turtle records its arched path −− contrast to the
;; effect of "pen−down"
```

---

standard−deviation

standard−deviation *list*

Reports the unbiased statistical standard deviation of a *list* of numbers. Ignores other types of items.

```
show standard−deviation [1 2 3 4 5 6]
=> 1.8708286933869707
show standard−deviation values−from turtles [energy]
;; prints the standard deviation of the variable "energy"
;; from all the turtles
```

startup

startup

User−defined procedure which, if it exists, will be called when a model is first loaded.

```
to startup
  setup
end
```

stop

stop

The calling agent exits immediately from the enclosing procedure, ask, or ask−like construct (cct, hatch, sprout). Only the current procedure stops, not all execution for the agent.

Note: stop can be used to stop a forever button. If the forever button directly calls a procedure, then when that procedure stops, the button stops. (In a turtle or patch forever button, the button won't stop until every turtle or patch stops −− a single turtle or patch doesn't have the power to stop the whole button.)

substring

substring *string position1 position2*

Reports just a section of the given string, ranging between the given positions.

Note: The positions are numbered beginning with 0, not with 1.

```
show substring "turtle" 1 4
=> "urt"
```

sum

sum *list*

Reports the sum of the elements in the list.

```
show sum values−from turtles [energy]
;; prints the total of the variable "energy"
```

```
;; from all the turtles
```

T

tan

tan *number*

Reports the tangent of the given angle. Assumes the angle is given in degrees.

timer

timer

Reports the current value of the global clock since the command reset−timer was last run. The resolution of the clock is milliseconds. (Whether it is accurate to the nearest millisecond may vary from system to system.)

to

to *procedure−name*
to *procedure−name* [*input1 input2* ...]

Used to begin a command procedure.

```
to setup
  ca
  crt 500
end

to circle [radius]
  cct 100 [ fd radius ]
end
```

to−report

to−report *procedure−name*
to−report *procedure−name* [*input1 input2* ...]

Used to begin a reporter procedure.

The body of the procedure should use `report` to report a value for the procedure. See report.

```
to−report average [a b]
  report (a + b) / 2
end

to−report absolute−value [number]
  ifelse number >= 0
    [ report number ]
    [ report 0 − number ]
end
```

```
to-report first-turtle?
  report who = 0  ;; reports true or false
end
```

towards
towards-nowrap

towards *agent*
towards-nowrap *agent*


Reports the heading from this agent to the given agent.

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, towards will report the heading of the wrapped path. towards-nowrap never uses the wrapped path.

towardsxy
towardsxy-nowrap

towardsxy *x y*
towardsxy-nowrap *x y*


Reports the heading from the turtle or patch towards the point (*x*,*y*).

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, towardsxy will report the heading of the wrapped path. towardsxy-nowrap never uses the wrapped path.

turtle

turtle *number*

Reports the turtle with the given ID number, or nobody if there is no such turtle. *number* must be an integer.

```
set color-of turtle 5 red
;; turtle with id number 5 turns red
ask turtle 5 [ set color red ]
;; another way to do the same thing
```

turtles

turtles

Reports the agentset consisting of all turtles.

```
show count turtles
;; prints the number of turtles
```

turtles−at
BREED−at

turtles−at *dx dy*
BREED*−at *dx dy

Reports an agentset containing the turtles on the patch (dx, dy) from the caller (including the caller itself if it's a turtle). If the caller is the observer, dx and dy are calculated from the origin (0,0).

```
;; suppose I have 40 turtles at the origin
show count turtles−at 0 0
=> 40
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are counted.

```
breeds [cats dogs]
create−custom−dogs 5 [ setxy 2 3 ]
show count dogs−at 2 3
=> 5
```

---

turtles−here
BREED−here

turtles−here
***BREED*−here**

Reports an agentset containing all the turtles on the caller's patch (including the caller itself if it's a turtle).

```
ca
crt 10
ask turtle 0 [ show count turtles−here ]
=> 10
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are counted.

```
breeds [cats dogs]
create−cats 5
create−dogs 1
ask dogs [ show count cats−here ]
=> 5
```

See also other−turtles−here.

---

turtles−own
BREED−own

turtles−own [*var1 var2* ...]
***BREED*−own [*var1 var2* ...]**

The turtles−own keyword, like the globals, breed, <*BREED*>−own, and patches−own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables

belonging to each turtle.

If you specify a breed instead of "turtles", only turtles of that breed have the listed variables. (More than one breed may list the same variable.)

```
breeds [cats dogs hamsters]
turtles-own [eyes legs]   ;; applies to all breeds
cats-own [fur kittens]
hamsters-own [fur cage]
dogs-own [hair puppies]
```

See also globals, patches-own, breeds, *<BREED>*-own.

---

type

type *value*

Prints *value* in the Command Center, *not* followed by a carriage return (unlike print and show). The lack of a carriage return allows you to print several values on the same line.

The calling agent is *not* printed before the value. unlike show.

```
type 3 type " " print 4
=> 3 4
```

See also print, show.

U

---

uphill

uphill *patch-variable*

Reports the turtle heading (between 0 and 359 degrees) in the direction of the maximum value of the variable *patch-variable*, of the patches in a one-patch radius of the turtle. (This could be as many as eight or as few as five patches, depending on the position of the turtle within its patch.)

If there are multiple patches that have the same greatest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need to reach the nearest corner of that patch is reported.

See also uphill4, downhill, downhill.

---

uphill4

uphill4 *patch−variable*

Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the maximum value of the variable *patch−variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same greatest value, a random patch from those patches will be selected.

See also uphill, downhill, downhill4.

---

user−choice

user−choice *value list−of−choices*

Opens a dialog with *value* displayed as the message and a button corresponding to each item in *list−of−choices*.

Reports the item in *list−of−choices* that is associated with the button the user presses.

value may be of any type, but is typically a string.

```
if "yes" = (user−choice "Do you really want to setup the model?" [ "no" "yes" ])
  [ setup ]
```

---

user−input

user−input *value*

Reports the string that a user types into a textbox in a dialog with title *value*.

The value may be of any type, but is typically a string.

```
show user−input "What is your name?"
```

---

user−message

user−message *value*

Opens a dialog with *value* displayed as the message.

The value may be of any type, but is typically a string.

```
user−message "There are " + count turtles + " turtles."
```

V

---

value−from

value−from *agent* [*reporter*]

Reports the value of the reporter for the given agent (turtle or patch).

```
show value−from (turtle 5) [who * who]
=> 25
show value−from (patch 0 0) [count turtles in−radius 3]
;; prints the number of turtles located within a
;; three−patch radius of the origin
```

values−from

values−from *agentset* [*reporter*]

Reports a list that contains the value of the reporter for each agent in the agentset.

```
ca
crt 4
show values−from turtles [who]
=> [0 1 2 3]
show values−from turtles [who * who]
=> [0 1 4 9]
```

variance

variance *list*

Reports the unbiased statistical variance of a *list* of numbers. Ignores other types of items.

```
show variance [2 7 4 3 5]
=> 3.7
```

W

wait

wait *number*

Wait the given number of seconds. (You can use floating−point numbers to specify fractions of seconds.) Note that you can't expect complete precision; the agent will never wait less than the given amount, but might wait slightly more.

```
repeat 10 [ fd 1 wait 0.5 ]
```

See also every.

while

while [*reporter*] [*commands*]

If *reporter* reports false, exit the loop. Otherwise run *commands* and repeat.

The reporter may have different values for different agents, so some agents may run *commands* a different number of times than other agents.

```
while [any other−turtles−here]
  [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

with

***agentset* with [*reporter*]**

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a boolean reporter. Reports a new agentset containing only those agents that reported true −− in other words, the agents satisfying the given condition.

```
show count patches with [pcolor = red]
;; prints the number of red patches
```

without−interruption

without−interruption [*commands*]

The agent runs all the commands in the block without allowing other agents to "interrupt". That is, other agents are put "on hold" and do not execute any commands until the commands in the block are finished.

Note: you may not use without−interruption in a context where the code is already running uninterrupted. (Other commands that establish such a context, besides without−interruption itself, are hatch, sprout, and create−custom−turtles.) We plan to lift this restriction in a future version of NetLogo.

```
crt 5
ask turtles
  [ without−interruption
      [ type 1 fd 1 type 2 ] ]
=> 1212121212
;; because each turtle will output 1 and move, then output 2
;; however:
ask turtles
  [ type 1 fd 1 type 2 ]
=> 1111122222
;; because each turtle will output 1 and move, then output 2
```

word

word *value1 value2*

Concatenates the two inputs together and reports the result as a string.

```
show word "tur" "tle"
=> "turtle"
word "a" 6
=> "a6"
set directory "c:\\foo\\fish\\"
show word directory "bar.txt"
=> "c:\foo\fish\bar.txt"
show word [1 54 8] "fishy"
=> "[1 54 8]fishy"
```

---

wrap−color

wrap−color *number*

wrap−color checks whether *number* is in the NetLogo color range of 0 to 140 (not including 140 itself). If it is not, wrap−color "wraps" the numeric input to the 0 to 140 range.

The wrapping is done by repeatedly adding or subtracting 140 from the given number until it is in the 0 to 140 range. (This is the same wrapping that is done automatically if you assign an out−of−range number to the color turtle variable or pcolor patch variable.)

```
show wrap−color 150
=> 10
show wrap−color −10
=> 130
```

X

---

xor

boolean1* xor *boolean2

Reports true if either *boolean1* or *boolean2* is true, but not when both are true.

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper−left and lower−right quadrants turn blue
```